Audit Report

Astroport Transmuter Pool

v1.0

January 30, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

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

Introduction

Purpose of This Report

Oak Security has been engaged by Astroport Protocol Foundation to perform a security audit of the Astroport Transmuter Pool.

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	`fc0c427d65690b56e9e9345f6156e48fb4e705db`
Scope	The scope was restricted to: ● Changes since our previous audit performed at commit `e565c3956a07b273b7bef7b0cd7877c24762000a` of the following directories: ○ `contracts/factory/*` ○ `packages/astroport/src/factory.rs` ● Full audit of the following directory: ○ `contracts/pair_transmuter/*`

Fixes verified at commit	`d9552605c4573ab0b1ff1524762782456265e9ef` Note that changes to the codebase beyond fixes after the initial audit have not been in the scope of our fixes review.

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

The Astroport Transmuter Pool implements a constant sum pair that supports swapping assets at a 1:1 ratio without incurring spread and fees.

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**, 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	Low	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	Documentation is available at `contracts/pair_transmuter/README.md`.
Test coverage	Medium-High	`cargo tarpaulin` reports coverage of `79.04%`.

Summary of Findings

No	Description	Severity	Status
1	Attackers can drain funds from the pair by providing worthless assets	Critical	Resolved
2	Pools can support an arbitrary number of assets	Minor	Resolved
3	Withdrawing liquidity from a pool with an empty side can lead to panic	Informational	Resolved
4	Liquidity pool token contracts use the same symbol	Informational	Acknowledged
5	Unused fields in `ProvideLiquidity` and `Swap` messages	Informational	Partially Resolved
6	Pair configuration override due to custom pair types	Informational	Acknowledged
7	Migrated pair configurations can be instantiated by anyone	Informational	Acknowledged

Detailed Findings

1. Attackers can drain funds from the pair by providing worthless assets

Severity: Critical

In `contracts/pair_transmuter/src/contract.rs:312-315`, the `swap` function accepts the `offer_asset` provided by the caller in return for other assets available in the pool.

However, no validation ensures that the provided asset is one of the pool's supported assets in `config.pair_info.asset_infos`.

This is problematic because attackers can offer any worthless native token in exchange for valuable assets in the pool, thereby profiting from the price difference and causing a loss of funds for liquidity providers.

Additionally, the `assert_sent_native_token_balance` function is designed not to validate CW20 tokens sent by the user (see `packages/astroport/src/asset.rs:248`). In contrast to the previous scenario, where an attacker supplies worthless native tokens, the attacker can simply specify CW20 tokens as the `offer_asset` to steal assets directly from the contract without providing actual funds.

Please see the `test_drain_pool` test case in the appendix to reproduce this issue.

Recommendation

We recommend implementing a check to ensure that the provided `offer_asset` is one of the pair assets in `config.pair_info.asset_infos`.

Status: Resolved

2. Pools can support an arbitrary number of assets

Severity: Minor

In `contracts/pair_transmuter/src/contract.rs:39-42`, the `instantiate` function ensures that at least two assets are submitted. However, no maximum amount of assets is enforced.

If the pool is configured to hold a very large number of assets, subsequent code that iterates over all assets leads to high gas expenses and potentially an out-of-gas error. An example of an iteration can be found in `contracts/pair_transmuter/src/contract.rs:184` when withdrawing liquidity.

We classify this issue as minor because it can only caused by the contract owner, which is a privileged address. Additionally, the contract is still usable even though the asset iteration fails due to an out-of-gas error. It allows users to provide single-sided liquidity, perform imbalance withdrawals, and explicitly specify the return asset during swaps. Since these features do not require iteration over all pool assets, an out-of-gas error will not occur.

Recommendation

We recommend enforcing a maximum amount of assets that can be supported.

Status: Resolved

3. Withdrawing liquidity from a pool with an empty side can lead to panic

Severity: Informational

The `withdraw_liquidity` function defined in `contracts/pair_transmuter/src/contract.rs:165-255` allows users to burn their liquidity pool tokens to withdraw the underlying pool liquidity.

However, if the `assets` vector is not explicitly specified and one of the pool's assets is empty, the `get_share_in_assets` function will include a zero-amount return asset in `contracts/pair_transmuter/src/utils.rs:50`.

Consequently, the transaction will fail when calling `BankMsg::Send` because Cosmos SDK does not allow zero-amount native token transfers.

Please see the `test_unbalanced_withdraw` test case to reproduce this issue.

Recommendation

We recommend filtering zero-amount assets in the `get_share_in_assets` function.

Status: Resolved

4. Liquidity pool token contracts use the same symbol

Severity: Informational

In `contracts/pair_transmuter/src/contract.rs:68`, the token symbol is hardcoded to `uLP` when instantiating a CW20 liquidity pool token contract. This causes all pair contracts to have the same symbol in their liquidity pool token contract.

Recommendation

We recommend dynamically deriving the token symbol from the `token_name`.

Status: Acknowledged

The client states that they don't think it is possible to craft anything user-friendly from, for example, IBC and token factory denoms, as the [symbol field in CW20 supports up to 11 characters](https://).

5. Unused fields in `ProvideLiquidity` and `Swap` messages

Severity: Informational

The `ProvideLiquidity` message is defined in `packages/astroport/src/pair.rs:42-51` as a struct with the following fields: `assets`, `slippage_tolerance`, `auto_stake`, and `receiver`.

However, the `slippage_tolerance` and `auto_stake` parameters are disregarded and unused in `contracts/pair_transmuter/src/contract.rs:123-125`.

Similarly, the `Swap` message is defined in `packages/astroport/src/pair.rs:53-59` as a struct with the following fields: `offer_asset`, `ask_asset_info`, `belief_price`, `max_spread`, and `to`.

However, the `belief_price` and `max_spread` parameters are disregarded and unused in `contracts/pair_transmuter/src/contract.rs:126-131`.

Unused parameters mislead users and decrease the user experience. For example, liquidity providers expect their liquidity pool tokens to be auto-staked when specifying the `auto_stake` parameter as `true`, which is incorrect because the field is ignored.

Recommendation

We recommend implementing messages specific to this contract by removing unused fields or returning an error if additional fields are provided.

Status: Partially Resolved

This issue is partially resolved because the `ProvideLiquidity` message will now return an error if `auto_stake` is specified as `Some(true)`. The client states that due to the `pair_transmuter` being one more pair type in Astroport, they keep the same APIs across their codebase (both smart contracts and UI).

6. Pair configuration override due to custom pair types

Severity: Informational

In `contracts/factory/src/migration.rs:45-46`, the `migrate_pair_configs` function migrates the `OldPairType::Concentrated` and `OldPairType::Custom` state into `PairType::Custom`.

Assume a scenario where `OldPairType::Concentrated` and `OldPairType::Custom("concentrated")` is specified. In this case, the concentrated pair type configuration will be overwritten by the custom pair type configuration, as seen in lines `61` and `63`.

We classify this issue as informational because it can only be caused by the contract owner, which is a privileged role.

Recommendation

We recommend revising the implementation to decide whether the intended design is to use `OldPairType::Custom("concentrated")` instead of `OldPairType::Concentrated` configuration.

Status: Acknowledged

The client states that the canonical enum for the passive concentrated liquidity pool is `PairType::Custom("concentrated")`.

7. Migrated pair configurations can be instantiated by anyone

Severity: Informational

In `contracts/factory/src/migration.rs:56`, the `migrate_pair_configs` function migrates previous pair configurations and sets the `permissioned` field to `false`. This allows existing pair configurations to be instantiated by anyone, as seen in `contracts/factory/src/contract.rs:295-297`.

We classify this issue as informational because it is a best practice to set default values as restrictive as possible, adhering to the principle of least privilege.

Recommendation

We recommend setting the `permissioned` field to `true`, explicitly requiring the contract owner to manually update the configurations for permissionless pools.

Status: Acknowledged

The client states that Astroport currently has only permissionless pair types. Hence, migration with "pemissioned: false" does not harm the protocol.

Appendix

1. Test case for "Attackers can drain funds from the pair by specifying CW20 assets"

Please run the test case in `contracts/pair_transmuter/tests/transmuter_integration.rs`.

```rust
#[test]
fn test_drain_pool() {
	let owner = Addr::unchecked("owner");

	let test_coins = vec![TestCoin::native("usdt"),
TestCoin::native("usdc")];

	let mut helper = Helper::new(&owner, test_coins.clone()).unwrap();

	helper
	.provide_liquidity(
		&owner,
		&[
		helper.assets[&test_coins[0]].with_balance(100_000_000000u128),
		helper.assets[&test_coins[1]].with_balance(100_000_000000u128),
		],
	)
	.unwrap();

	let user = Addr::unchecked("user");

	// 1. take all test_coins[0]
	let swap_asset = Asset {
	info: AssetInfo::Token { contract_addr: Addr::unchecked("astro") },
	amount: Uint128::new(100_000_000000u128)
	};
	helper.app.execute_contract(
	user.clone(),
	helper.pair_addr.clone(),
	&ExecuteMsg::Swap {
		offer_asset: swap_asset,
		ask_asset_info: None,
		belief_price: None,
		max_spread: None,
		to: None
	}, &[]
	).unwrap();

	let pool_info = helper.query_pool().unwrap();
	assert_eq!(
pool_info.assets,
vec![
	helper.assets[&test_coins[0]].with_balance(0u128),
	helper.assets[&test_coins[1]].with_balance(100_000_000000u128),
]
);

// 2. swap test_coins[0] for test_coins[1]
let swap_asset = Asset {
info: AssetInfo::NativeToken { denom: test_coins[0].denom().unwrap() },
amount: Uint128::new(100_000_000000u128)
};
helper.swap(&user, &swap_asset, None, None).unwrap();

let pool_info = helper.query_pool().unwrap();
assert_eq!(
pool_info.assets,
vec![
	helper.assets[&test_coins[0]].with_balance(100_000_000000u128),
	helper.assets[&test_coins[1]].with_balance(0u128),
]
);

// 3. take all test_coins[0]
let swap_asset = Asset {
info: AssetInfo::Token { contract_addr: Addr::unchecked("astro") },
amount: Uint128::new(100_000_000000u128)
};
helper.app.execute_contract(
user.clone(),
helper.pair_addr.clone(),
&ExecuteMsg::Swap {
	offer_asset: swap_asset,
	ask_asset_info: None,
	belief_price: None,
	max_spread: None,
	to: None
}, &[]
).unwrap();

// 4. empty pool
let pool_info = helper.query_pool().unwrap();
assert_eq!(
pool_info.assets,
vec![
	helper.assets[&test_coins[0]].with_balance(0u128),
	helper.assets[&test_coins[1]].with_balance(0u128),
]
);
```

```
}
```

2. Test case for "Withdrawing liquidity from an unbalanced pool can lead to panic"

Please run the test case in `contracts/pair_transmuter/tests/transmuter_integration.rs`.

```
#[test]
fn test_unbalanced_withdraw() {
       let owner = Addr::unchecked("owner");

       let test_coins = vec![TestCoin::native("usdt"),
TestCoin::native("usdc")];

       let mut helper = Helper::new(&owner, test_coins.clone()).unwrap();

       let user = Addr::unchecked("user");
       let provide_assets = [
       helper.assets[&test_coins[0]].with_balance(100_000_000000u128),
       helper.assets[&test_coins[1]].with_balance(100_000_000000u128),
       ];

       helper.give_me_money(&provide_assets, &user);

       helper.provide_liquidity(&user, &provide_assets).unwrap();

       let lp_balance = helper.token_balance(&helper.lp_token, &user);
       assert_eq!(lp_balance, 200_000_000000u128);

       // withdraw imbalanced
       helper
       .withdraw_liquidity(&user, 100_000_000000u128,
vec![helper.assets[&test_coins[0]].with_balance(100_000_000000u128)])
       .unwrap();

       let lp_balance = helper.token_balance(&helper.lp_token, &user);
       assert_eq!(lp_balance, 100_000_000000u128);

       let pool_info = helper.query_pool().unwrap();
       assert_eq!(
       pool_info.assets,
       vec![
              helper.assets[&test_coins[0]].with_balance(0u128),
              helper.assets[&test_coins[1]].with_balance(100_000_000000u128),
       ]
       );

       assert_eq!(
       helper.coin_balance(&test_coins[0], &user),
       100_000_000000u128
       );
```

```
        assert_eq!(
        helper.coin_balance(&test_coins[1], &user),
        0u128
        );

        // withdraw balanced, fails due to 0 native amount transacted
        helper
        .withdraw_liquidity(
               &user,
               100_000_000000u128,
               vec![],
        )
        .unwrap();
}
```