Audit Report

Astroport Incentives

v1.0

January 11, 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 Incentives smart contract.

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	`2beaf71a99b7d4826c2fb14076702794939df85f`
Scope	The following files were in the scope of the audit: ● `contracts/tokenomics/incentives/*` ● `packages/astroport/src/incentives.rs` ● The `determine_asset_info` function in `packages/astroport/src/asset.rs`
Fixes verified at commit	`9d30e0c6d1f4c878ad514089f8e993c4d4a70447` 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 Incentives contract is a reworked version of the Astroport Generator contract. It allocates token rewards for various liquidity tokens and distributes them pro-rata to liquidity stakers while supporting both CW20 and native tokens.

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	Medium-High	Complex reward distribution mechanisms are implemented along with multiple code transformations and iterations.
Code readability and clarity	Medium	Variable naming can be improved by explicitly specifying the context and purpose, such as renaming `next_update_ts` to `current_next_update_ts` in `contracts/tokenomics/incentives/src/state.rs:108`.
Level of documentation	Medium-High	Detailed documentation with illustrations is provided in the README file.
Test coverage	Medium-High	-

Summary of Findings

No	Description	Severity	Status
1	Users can incentivize pools without supplying native funds	Critical	Resolved
2	Users can claim excess rewards by specifying duplicate liquidity tokens	Critical	Resolved
3	Possibly incorrect reward calculations for new reward schedules	Major	Resolved
4	Malicious CW20 tokens can prevent legitimate rewards from being incentivized	Major	Resolved
5	External reward incentivization can be maliciously blocked	Minor	Partially Resolved
6	Native token factory liquidity tokens cannot be supported	Minor	Resolved
7	Updating blocked tokens does not affect existing reward tokens	Minor	Acknowledged
8	The factory contract cannot deregister pairs not stored in the `incentives` contract	Minor	Resolved
9	The remaining reward funds could be stuck after the owner deregisters them	Minor	Acknowledged
10	Orphaned rewards will be stuck in the contract after the schedule finishes	Minor	Resolved
11	Malicious liquidity tokens can bypass factory contract registration validation	Minor	Resolved
12	Typographic error in the codebase	Informational	Resolved
13	Lack of duplicate validation when updating blocked pool tokens	Informational	Resolved
14	Error message differs from implementation	Informational	Resolved
15	New pools can have zero allocation points	Informational	Resolved

Detailed Findings

1. Users can incentivize pools without supplying native funds

Severity: Critical

The `incentive` function in `contracts/tokenomics/incentives/src/utils.rs:285` calls the `assert_coins_properly_sent` function to determine that the funds sent with the message match the specified reward tokens. However, the `assert_coins_properly_sent` function does not perform the fund assertion if the supplied funds are empty, as seen in `packages/astroport/src/asset.rs:254`.

Consequently, users can incentivize native reward tokens without supplying them inside `info.funds`.

Please refer to the test_incentive_without_funds test case in the appendix to reproduce this issue.

Recommendation

We recommend modifying the `assert_coins_properly_sent` function to error if empty funds are provided.

Status: Resolved

2. Users can claim excess rewards by specifying duplicate liquidity tokens

Severity: Critical

In `contracts/tokenomics/incentives/src/execute.rs:37`, the `lp_tokens` vector specified by the user will be used to claim rewards from the pool. However, specifying duplicate liquidity tokens allows the user to claim more rewards as the existing pool and user positions are collected in a batch, causing the `claim_rewards` function to compute eligible rewards from an outdated state.

Consequently, users can claim excess rewards from the pool, causing a loss of funds for the protocol.

Please refer to the test_claim_excess_rewards test case in the appendix to reproduce this issue.

Recommendation

We recommend implementing validation to ensure no duplicate liquidity tokens are specified in the `lp_tokens` vector.

Status: Resolved

3. Possibly incorrect reward calculations for new reward schedules

Severity: Major

In `contracts/tokenomics/incentives/src/state.rs:59-61`, the `calculate_reward` function computes the user rewards by multiplying the global index and the user-bonded liquidity token amount if the user index is larger than the global index.

This happens because if a new reward schedule is added after the old reward schedule is completed, full rewards are accrued to the user as their liquidity token staking spans across both reward periods.

However, suppose the new reward schedule's global index is larger than the user index (e.g., due to a large reward amount or a smaller number of stakers). In that case, the rewards will be computed incorrectly in line `63`. Specifically, the reward is calculated by deducting the global and user indexes and multiplying them with the user-bonded liquidity token amount.

Consequently, users will not receive the total rewards they have staked across both periods, causing a loss of rewards for them.

Please refer to the test_user_claim_less test case in the appendix to reproduce this issue.

Recommendation

We recommend handling the case when the new reward schedule's global index is larger than the old schedule's user index to fix this issue.

Status: Resolved

4. Malicious CW20 tokens can prevent legitimate rewards from being incentivized

Severity: Major

In `contracts/tokenomics/incentives/src/utils.rs:325-329`, the `remove_reward_from_pool` function calls `into_msg` to transfer the remaining CW20 token rewards to the recipient. Internally, the `into_msg` function calls the `Cw20ExecuteMsg::Transfer` message on the CW20 token contract to complete the fund transfer.

The issue is that malicious CW20 tokens can revert on purpose when the `Transfer` message is called, causing the transaction to fail and preventing the owner from removing them. As the maximum number of external rewards allowed is five (see `contracts/tokenomics/incentives/src/state.rs:267`), legitimate reward tokens might not be possible to be added. Stakers would receive worthless tokens as rewards and would be disincentivized from providing liquidity to the pair contract.

Recommendation

We recommend dispatching the `Transfer` message as a `ReplyOn::Always` submessage with a dedicated reply handler. If the malicious CW20 token reverts on purpose, the reply handler can ignore the error so the owner removes the malicious token successfully and adds it to the list of `BLOCKED_TOKENS`.

Status: Resolved

5. External reward incentivization can be maliciously blocked

Severity: Minor

Reward incentivization is controlled through multiple factors, such as ensuring the reward is not in `BLOCKED_TOKENS` and requiring the caller to pay the `incentivization_fee_info` fee in `contracts/tokenomics/incentives/src/utils.rs:248`.

In `contracts/tokenomics/incentives/src/state.rs:267`, `MAX_REWARD_TOKENS` defines a maximum number of external reward token denoms that can be incentivized for a pair, which is a constant defined as five. Once this limit is reached, a pool can no longer be further incentivized through external tokens. This presents an opportunity for an attacker to incentivize a pool with multiple worthless non-blacklisted denoms in order to reach the limit and block legitimate external rewards incentivization. Although the `incentivization_fee_info` fee is implemented, a motivated attacker could block external rewards to a pool relatively inexpensively.

Additionally, the `BLOCKED_TOKENS` is configured as a vector that will increase in size for every blocked token. If the owner keeps adding the attacker's tokens to the list with the `UpdateBlockedTokenslist` message, an out-of-gas error might occur when `blocked_tokens.contains` is called, as the `contains` function operates with `O(n)` complexity.

We classify this issue as minor due to the mitigating effect of the incentivization fee.

Recommendation

We recommend implementing a whitelist mapping to store approved token denoms. While a whitelist may have more management overhead, it will provide more comprehensive coverage to block the scenario described above.

Alternatively, we recommend modifying the storage design of BLOCKED_TOKENS to use a mapping so the complexity can be reduced. This can be accomplished by accessing the mapping key instead of iterating over all blocked tokens. However, this would only partially fix the issue as it does not fully resolve the concern about a motivated actor spamming external rewards tokens.

Status: Partially Resolved

This issue is partially resolved through the storage design of BLOCKED_TOKENS being modified to use a mapping. The client states that the incentives contract is meant to minimize governance management of reward flows. They consider fees as a sustainable barrier to prevent malicious reward spam.

6. Native token factory liquidity tokens cannot be supported

Severity: Minor

In contracts/tokenomics/incentives/src/utils.rs:346-349, the query_pair_info function calls the pair_info_by_pool function after parsing the lp_minter address from the native token factory denom. The parsed address will be the pair contract address because the second substring in token factory denoms represents the creator's address, which is authorized to mint and burn tokens.

The problem occurs when the pair_info_by_pool function in packages/astroport/src/asset.rs:656 attempts to retrieve the pair contract address by sending a Cw20QueryMsg::Minter query message on the lp_minter address.

This implies that calling the pair_info_by_pool function for token factory denoms will fail because the Minter query is unavailable in a pair contract.

We classify this issue as minor because the support for pair contracts to use token factory denoms is not implemented yet at the audited commit.

Recommendation

We recommend performing a PairQueryMsg::Pair query message on the parsed lp_minter address directly to retrieve the pair information.

Status: Resolved

7. Updating blocked tokens does not affect existing reward tokens

Severity: Minor

The update_blocked_pool_tokens function in contracts/tokenomics/incentives/src/execute.rs:428-503 automatically removes active pools that contain assets that are part of the newly added `BLOCKED_TOKENS`. However, this is not enforced for existing reward tokens.

Specifically, the `incentivize` function does not allow blocked tokens to be added as an external reward in `contracts/tokenomics/incentives/src/utils.rs:221`. As this is only validated when incentivizing new reward tokens, existing reward schedules that contain blocked tokens are not removed.

Therefore, although a blocked token will no longer be eligible for new rewards, any pool that already includes it will keep distributing the token.

Recommendation

We recommend implementing an entry point for the owner to remove existing rewards that are part of `BLOCKED_TOKENS` and withdraw them.

Status: Acknowledged

The client acknowledges this issue, stating this can be done by off-chain query `QueryMsg::BlockedTokensList{}` and direct call to `ExecuteMsg::RemoveRewardFromPool{}`.

8. The factory contract cannot deregister pairs not stored in the `incentives` contract

Severity: Minor

In `contracts/tokenomics/incentives/src/utils.rs:110`, the `deactivate_pool` function calls `PoolInfo::load` when the factory contract deregisters a pair contract address. Such transactions will fail and revert if the liquidity token is not stored in the incentives contract though.

This could happen if no users stake the liquidity token in the `incentives` contract or the liquidity token is not incentivized internally or externally.

Please refer to the test_cannot_deactivate_pool test case in the appendix to reproduce this issue.

Recommendation

We recommend using `PoolInfo::may_load` so the transaction will not fail if the liquidity token is not stored.

Status: Resolved

9. The remaining reward funds could be stuck after the owner deregisters them

Severity: Minor

The `RemoveRewardFromPool` message allows the owner to remove an external reward token and optionally remove upcoming scheduled rewards in `contracts/tokenomics/incentives/src/state.rs:380`. If the owner does not choose to remove upcoming rewards from the `EXTERNAL_REWARD_SCHEDULES` storage, the funds will remain in the contract and will only be distributed if someone incentivizes it for the liquidity pool.

However, if the reward schedule period has ended and the reward is not incentivized, the funds will be stuck in the contract because there is no entry point for the owner to withdraw them.

We classify this issue as minor because it can only be caused by the owner passing a `true` value for the `bypass_upcoming_schedules` argument.

Recommendation

We recommend implementing an entry point for the owner to remove outdated `EXTERNAL_REWARD_SCHEDULES` and withdraw the remaining funds.

Status: Acknowledged

The client acknowledges there is a risk that rewards cannot be withdrawn from the contract if `bypass_upcoming_schedules` is `true` when deregistering a reward from a pool.

The `bypass_upcoming_schedules` switch is added for emergency cases, which is discouraged from being used. For example, suppose governance cannot deregister a reward, and it prevents another valuable token from entering the rewards bucket. In that case, governance can remove it from the bucket, leaving outstanding tokens in the generator balance forever.

Additionally, the Astroport DAO will be the only owner of the `incentives` contract, and according to their limitations (max 25 upcoming periods) and due to a small size of data (just one `Decimal`) stored in the `EXTERNAL_REWARD_SCHEDULES` state, it is highly unlikely they hit the gas limit when they intend not to bypass upcoming schedules (i.e., `bypass_upcoming_schedules` is `false`).

10. Orphaned rewards will be stuck in the contract after the schedule finishes

Severity: Minor

In `contracts/tokenomics/incentives/src/state.rs:158`, the rewards are added to the orphaned rewards if there are no liquidity token stakers. An edge case scenario is that if there are no liquidity tokens staked after the reward ends, the global index will remain zero when added into `FINISHED_REWARD_INDEXES`, and the orphaned rewards will be stuck in the contract.

We classify this issue as minor because users are expected to be incentivized to stake their liquidity tokens for rewards.

Recommendation

We recommend implementing an entry point for the contract owner to withdraw the orphaned rewards in case the above edge case happens.

Status: Resolved

11. Malicious liquidity tokens can bypass factory contract registration validation

Severity: Minor

In `contracts/tokenomics/incentives/src/utils.rs:379`, the `is_pool_registered` function compares the pair contract address queried from the factory contract (`factory::QueryMsg::Pair`) to match the pair contract address queried from the liquidity token (`PairQueryMsg::Pair`). This is used to ensure the liquidity token is not forged and originates from a pair contract instantiated by Astroport's factory contract.

However, this validation can be bypassed by creating a malicious CW20 token that returns `PairInfo.contract_addr` as a legitimate pair contract address registered in the factory contract. This is because the validation relies on a response returned by the untrusted contract, which can be manipulated.

Consequently, the liquidity token validation in `deposit`, `setup_pools`, and `incentivize` functions can be bypassed, allowing untrusted liquidity tokens to receive reward distributions.

Recommendation

We recommend validating the `PairInfo.liquidity_token` queried from the factory contract to match the supplied liquidity token address.

Status: Resolved

12. Typographic error in the codebase

Severity: Informational

In `packages/astroport/src/incentives.rs:27`, there is a spelling error. `insentivization_fee_info` should be spelled as `incentivization_fee_info`. It is best practice to resolve variable name spelling errors to improve the readability of the codebase.

Recommendation

We recommend updating the spelling error as mentioned above.

Status: Resolved

13. Lack of duplicate validation when updating blocked pool tokens

Severity: Informational

The `update_blocked_pool_tokens` function in `packages/astroport/src/execute.rs:398` does not validate if there are duplicates between the `add` and the `remove` input vectors. Given that the `remove` operation is done first in lines `414-425` followed by the `add` operation, in case an asset is present in both, it will end up being added to `BLOCKED_TOKENS` without further warning.

Recommendation

We recommend ensuring there are no duplicated addresses between both input vectors.

Status: Resolved

14. Error message differs from implementation

Severity: Informational

In `packages/astroport/src/incentives.rs:52-55`, the number of periods of a schedule is validated. The `if` statement allows that `input.duration_periods == MAX_PERIODS`; however, the error message states, "`Duration must be more 0 and less than {MAX_PERIODS}`", which differs from the implementation.

Recommendation

We recommend modifying the error message so it reflects the implementation correctly.

Status: Resolved

15. New pools can have zero allocation points

Severity: Informational

The `setup_pools` function in `contracts/tokenomics/incentives/src/execute.rs:241` allows allocating zero `alloc_points` to any given pool. This will result in the pool not receiving any ASTRO rewards.

Recommendation

We recommend validating that the amount of `alloc_point` of any given pool is larger than zero.

Status: Resolved

Appendix

1. Test case for “[Users can incentivize pools without supplying native funds]”

```
#[test]
fn test_incentive_without_funds() {
    // reproduced in
contracts/tokenomics/incentives/tests/incentives_integration_tests.rs
    let astro = native_asset_info("astro".to_string());
    let usdc =  native_asset_info("usdc".to_string());
    let mut helper = Helper::new("owner", &astro).unwrap();
    let owner = helper.owner.clone();

    let asset_infos = [AssetInfo::native("foo"), AssetInfo::native("bar")];
    let pair_info = helper.create_pair(&asset_infos).unwrap();
    let lp_token = pair_info.liquidity_token.to_string();

    let provide_assets = [
        asset_infos[0].with_balance(100000u64),
        asset_infos[1].with_balance(100000u64),
    ];
    // Owner provides liquidity first just make following calculations easier
    // since first depositor gets small cut of LP tokens
    helper
        .provide_liquidity(
            &owner,
            &provide_assets,
            &pair_info.contract_addr,
            false, // Owner doesn't stake in generator
        )
        .unwrap();

    let bank = TestAddr::new("bank");
    let reward_asset_info = usdc.clone();
    let reward = reward_asset_info.with_balance(1000_000000u128);
    helper.mint_assets(&bank, &[reward.clone()]);

    let (schedule, _) = helper.create_schedule(&reward, 2).unwrap();
    let incentivization_fee = helper.incentivization_fee.clone();
    helper.mint_coin(&bank, &incentivization_fee);

    // add reward
    helper.app.execute_contract(
        bank.clone(),
        helper.generator.clone(),
        &ExecuteMsg::Incentivize {
            lp_token: lp_token.to_string(),
            schedule,
        },
        &[incentivization_fee], // only send incentivization fee without reward
    ).unwrap();

}
```

2. Test case for "Users can claim excess rewards by specifying duplicate liquidity tokens"

```
#[test]
fn test_claim_excess_rewards() {
    // reproduced in
contracts/tokenomics/incentives/tests/incentives_integration_tests.rs
    let astro = native_asset_info("astro".to_string());
    let mut helper = Helper::new("owner", &astro).unwrap();
    let owner = helper.owner.clone();

    let mut pools = vec![
        ("uusd", "eur", "".to_string(), vec!["user1", "user2"], 100),
        ("uusd", "tokenA", "".to_string(), vec!["user1"], 50),
        ("uusd", "tokenB", "".to_string(), vec!["user2"], 50),
    ];

    let mut active_pools = vec![];
    for (token1, token2, lp_token, stakers, alloc_points) in pools.iter_mut() {
        let asset_infos = [AssetInfo::native(*token1),
AssetInfo::native(*token2)];
        let pair_info = helper.create_pair(&asset_infos).unwrap();
        *lp_token = pair_info.liquidity_token.to_string();
        active_pools.push((pair_info.liquidity_token.to_string(),
*alloc_points));

        let provide_assets = [
            asset_infos[0].with_balance(100000u64),
            asset_infos[1].with_balance(100000u64),
        ];
        // Owner provides liquidity first just make following calculations
easier
        // since first depositor gets small cut of LP tokens
        helper
            .provide_liquidity(
                &owner,
                &provide_assets,
                &pair_info.contract_addr,
                false, // Owner doesn't stake in generator
            )
            .unwrap();

        for staker in stakers {
            let staker_addr = TestAddr::new(staker);

            // Pool doesn't exist in Generator yet
            let astro_before = astro.query_pool(&helper.app.wrap(),
&staker_addr).unwrap();
            helper
                .claim_rewards(&staker_addr,
vec![pair_info.liquidity_token.to_string(),
pair_info.liquidity_token.to_string()])
                .unwrap_err();
            let astro_after = astro.query_pool(&helper.app.wrap(),
&staker_addr).unwrap();
            assert_eq!((astro_after - astro_before).u128(), 0);

            helper
                .provide_liquidity(
                    &staker_addr,
                    &provide_assets,
                    &pair_info.contract_addr,
                    true,
                )
                .unwrap();
        }
    }

    helper.setup_pools(active_pools).unwrap();
    helper.set_tokens_per_second(1_000000).unwrap();

    helper
        .app
        .update_block(|block| block.time = block.time.plus_seconds(5));

    let user1 = TestAddr::new("user1");
    let astro_before = astro.query_pool(&helper.app.wrap(), &user1).unwrap();

    helper
        .claim_rewards(&user1, vec![
            pools[0].2.to_string(),
            pools[1].2.to_string(),
            pools[0].2.to_string(),
            pools[1].2.to_string(),
            ])
        .unwrap();
    let astro_after = astro.query_pool(&helper.app.wrap(), &user1).unwrap();
    assert_eq!((astro_after - astro_before).u128(), 2_500000); // users get more
rewards

}
```

3. Test case for “Possible incorrect reward calculations for new reward schedules”

```
#[test]
fn test_user_claim_less() {
    // reproduced in
contracts/tokenomics/incentives/tests/incentives_integration_tests.rs
    let astro = native_asset_info("astro".to_string());
    let mut helper = Helper::new("owner", &astro).unwrap();
    let owner = helper.owner.clone();
    let incentivization_fee = helper.incentivization_fee.clone();

    let asset_infos = [AssetInfo::native("foo"), AssetInfo::native("bar")];
    let pair_info = helper.create_pair(&asset_infos).unwrap();
    let lp_token = pair_info.liquidity_token.to_string();

    let provide_assets = [
        asset_infos[0].with_balance(100000u64),
        asset_infos[1].with_balance(100000u64),
    ];
    // Owner provides liquidity first just make following calculations easier
    // since first depositor gets small cut of LP tokens
    helper
        .provide_liquidity(
            &owner,
            &provide_assets,
            &pair_info.contract_addr,
            false, // Owner doesn't stake in generator
        )
        .unwrap();

    let user = TestAddr::new("user");
    helper
        .provide_liquidity(&user, &provide_assets, &pair_info.contract_addr,
true)
        .unwrap();

    let bank = TestAddr::new("bank");
    let reward_asset_info = AssetInfo::native("reward");
    let reward = reward_asset_info.with_balance(1000_000000u128);

    // create reward schedule
    helper.mint_assets(&bank, &[reward.clone()]);
    let (schedule, internal_sch) = helper.create_schedule(&reward, 2).unwrap();
    helper.mint_coin(&bank, &incentivization_fee);

    let additional_random_funds = coin(1000u128, "uusd");
    helper.mint_coin(&bank, &additional_random_funds);

    helper
```

```
        .incentivize(
            &bank,
            &lp_token,
            schedule.clone(),
            &[incentivization_fee.clone()],
        )
        .unwrap();

    helper.app.update_block(|block| {
        block.time = Timestamp::from_seconds(internal_sch.next_epoch_start_ts)
    });

    // user claim, sets user index
    helper.claim_rewards(&user, vec![lp_token.clone()]).unwrap();

    // finish 1st schedule, reward goes to FINISHED_REWARD_INDEXES
    helper.app.update_block(|block| {
        block.time = Timestamp::from_seconds(internal_sch.end_ts + 1)
    });

    // create reward schedule again
    helper.mint_assets(&bank, &[reward.clone()]);
    let (schedule, internal_sch) = helper.create_schedule(&reward, 2).unwrap();
    helper.mint_coin(&bank, &incentivization_fee);

    let additional_random_funds = coin(1000u128, "uusd");
    helper.mint_coin(&bank, &additional_random_funds);

    helper
        .incentivize(
            &bank,
            &lp_token,
            schedule.clone(),
            &[incentivization_fee.clone()],
        )
        .unwrap();

    // few seconds before schedule finishes
    helper.app.update_block(|block| {
        block.time = Timestamp::from_seconds(internal_sch.end_ts - 1)
    });

    // user claim rewards as (global index - user index), which is incorrect
    helper.claim_rewards(&user, vec![lp_token.clone()]).unwrap();

    // finish 2nd schedule
    helper.app.update_block(|block| {
        block.time = Timestamp::from_seconds(internal_sch.end_ts + 1)
    });
    // user claim all rewards
    helper.claim_rewards(&user, vec![lp_token.clone()]).unwrap();

    // check user rewards
    let new_reward_balance = reward_asset_info
            .query_pool(&helper.app.wrap(), &user)
            .unwrap();

    // did not claim full rewards even after rounding
    assert_eq!(new_reward_balance, reward.amount + reward.amount);

}
```

4. Test case for “The factory contract cannot deregister pairs not stored in the incentives contract”

```rust
#[test]
fn test_cannot_deactivate_pool() {
    // reproduced in
contracts/tokenomics/incentives/tests/incentives_integration_tests.rs
    let astro = native_asset_info("astro".to_string());
    let mut helper = Helper::new("owner", &astro).unwrap();

    let tokens = [
        AssetInfo::native("usd"),
        AssetInfo::native("foo"),
    ];

    let asset_info = &[tokens[0].clone(), tokens[1].clone()];

    // factory contract create pair
    helper
        .create_pair(asset_info)
        .unwrap();

    // ensure pair created
    let pair_info = helper.query_pair_info(asset_info);
    assert_eq!(pair_info.asset_infos, asset_info);

    // factory contract cannot deregister pair
    helper
        .deactivate_pool_full_flow(asset_info)
        .unwrap();

}
```