THREE SIGMA

IndexTokenStaking Staking

Security Review

Disclaimer

Security Review

IndexTokenStaking Staking

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

IndexTokenStaking Staking

Table of Contents

Summary

Security Review

IndexTokenStaking Staking

Summary

Three Sigma audited OpenDelta's Index Token Staking in a 2 person week engagement. The audit was conducted from 22/04/2025 to 28/04/2025.

Protocol Description

The Index Token Staking Program is a Solana smart contract that allows users to stake their index tokens and earn reward tokens through a time-based distribution system. It implements a complete staking lifecycle including initialization, deposits, staking/unstaking, and reward claims, while featuring dynamic reward calculations proportional to users' staked amounts and configurable distribution periods.

Scope

Security Review

IndexTokenStaking Staking

Scope

Filepath	nSLOC
programs/index_token_staking/src/instructions/create_staking_account.rs	48
programs/index_token_staking/src/instructions/deposit_reward_token.rs	117
programs/index_token_staking/src/instructions/initialize.rs	72
programs/index_token_staking/src/instructions/mod.rs	14
programs/index_token_staking/src/instructions/stake.rs	94
programs/index_token_staking/src/instructions/unstake.rs	61
programs/index_token_staking/src/instructions/withdraw_rewards.rs	92
programs/index_token_staking/src/instructions/withdraw_stake.rs	37
programs/index_token_staking/src/lib.rs	59
programs/index_token_staking/src/state.rs	89
programs/index_token_staking/src/utils.rs	17
Total	**700**

Methodology

Security Review

IndexTokenStaking Staking

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

IndexTokenStaking Staking

Project Dashboard

Application Summary

Name	Index Token Staking
Repository	https://github.com/OpenDeltaLabs/index_token_staking_audit
Commit	8385361
Language	Rust
Platform	Solana

Engagement Summary

Timeline	22/04/2025 to 28/04/2025
№ of Auditors	2
Review Time	2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	0	0	0
Medium	2	2	0
Low	6	6	0
None	5	5	0

Category Breakdown

Suggestion	5
Documentation	0
Bug	9
Optimization	0
Good Code Practices	0

Risk Section

Security Review

IndexTokenStaking Staking

Risk Section

- **Administrative Key Management Risks**: The system makes use of administrative keys to manage critical operations. Compromise or misuse of these keys could result in unauthorized actions and financial losses.

Findings

Security Review

IndexTokenStaking Staking

Findings

3S-IndexTokenStaking-C01

Multiple staking_states can be created for the same stake-mint, letting attackers steal rewards and principal

Id	3S-IndexTokenStaking-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in #df44d86.

Description

In IndexTokenStaking, the pool state (staking_state) is keyed by both the stake-token mint and the pool authority, so anyone can initialize parallel pools that accept the same token but track rewards separately. By contrast, a user's position (user_staking_account) is keyed only by the mint and the user's key, so that a single PDA is silently shared across every staking_state built on the same token.

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/instructions/stake.rs#L61C9-L61C98:

```
// staking_state-specific
seeds = [STAKING_STATE_SEED, stake_token_mint.key().as_ref(),
authority.key().as_ref()],
]
```

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/instructions/stake.rs#L40 :

```
// user position  (authority missing)
seeds = [USER_STAKING_ACCOUNT_SEED, stake_token_mint.key().as_ref(),
user.key().as_ref()],
```

]

Because deposits made in one staking_state increase user_staking_account.staked_amount for all staking_states while only the local total_staked is updated, an attacker can over-state their stake in a victim staking_state, drain its reward vault, and make staking_states insolvent.

If two states share the same stake mint, an attacker is able to

call unstake() in pool A (which internally uses the correct staking_state); then call withdraw_stake() but pass the vault of pool B.

Because withdraw_stake() does not verify that the supplied staking_state matches the vault it debits, the attacker successfully pulls tokens from B's vault, leaving pool A insolvent for honest stakers.

Steps to Reproduce

Reward-vault drainage:

1. Alice initializes pool *P_0* with authority = Alice.

2. Bob initializes pool *P_1* with the same stake_token_mint but authority = Bob.

3. Bob stakes N tokens in *P_1*; user_staking_account.staked_amount becomes N.

4. Bob calls claim_rewards/withdraw_rewards on *P_0*. *P_0* uses the bloated staked_amount with its smaller total_staked, over-paying Bob.

Cross-pool principal theft / Insolvency

1. Attacker calls **unstake** in pool **A**; A's **staking_state** is updated correctly.

2. Attacker calls **withdraw_stake** but supplies **B**'s vault and B-specific accounts.

3. Because **withdraw_stake** never checks that the vault matches the supplied **staking_state**, tokens are pulled from B's vault.

4. Pool A now shows the position closed while pool B has lost funds, leaving honest stakers under-collateralised.

Refer to the Proof of concept for a coded test.

Recommendation

Bind each user position to a single pool by including the the staking_state's key in the PDA seeds:

```
#[account(
        init,
        payer = user,
        space = 8 + std::mem::size_of::<UserStakingAccount>(),
        // the user staking account must encode the staked token mint and the users public
key.
        seeds = [USER_STAKING_ACCOUNT_SEED, stake_token_mint.key().as_ref(),
user.key().as_ref(),
+staking_state.key().as_ref(),
],
        bump
    )]
    pub user_staking_account: Account<'info, UserStakingAccount>,
```

3S-IndexTokenStaking-M01

Staking will get permanently DoS'ed if their total_staked goes down heavily and round finished

Id	3S-IndexTokenStaking-M01
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #e8b2e9c.

Description

When depositing rewards, we are checking that the **total_staked** value is greater than **base**. base is simply 1 unit value of the staked token.

deposit_reward_token.rs#L68-L71

```
pub fn handle_deposit_reward_token(ctx: Context<DepositRewardToken>, amount: u64) ->
Result<()> {
    ...
    require!(
        staking_state.total_staked as u128 >= base,
        CustomError::InsufficientStake
    );
    ...
}
```

For users to stake, we are preventing new stakers to stake, if the Round finished.

stake.rs#L92

```
pub fn handle_stake(ctx: Context<Stake>, amount: u64) -> Result<()> {
    ...
    // Don't allow staking when there is no time remaining after the first rewards have been
deposited.
    if end_staking_period != 0 {
        let time_remaining = end_staking_period
```

```
            .checked_sub(current_time)
            .ok_or(CustomError::TimeUnderflow)?;
>>      require!(time_remaining > 0, CustomError::StakingEnded);
    }
    ...
}
```

- No one can stake after the staking round finish

- The new round only starts after depositing tokens calling **handle_deposit_reward_token**

- **handle_deposit_reward_token** will not accept depositing tokens if **total_staked** is less than **base** (1 token unit)

- If at a given round a lot of stakers unstaked and **total_staked** goes below **base** (1 token unit), and round finished. The Protocol will get permanently DoS'ed

- No one can stake, admins can't deposit tokens, the Staking will be stopped forever.

Recommendations

Modify The check to only be for the first round only, if the round is not the first, don't enforce the check.

```
pub fn handle_deposit_reward_token(ctx: Context<DepositRewardToken>, amount: u64) ->
Result<()> {
    ...
    require!(
-       staking_state.total_staked as u128 >= base,
+        staking_state.total_staked as u128 >= base || staking_state.end_staking_period > 0,
        CustomError::InsufficientStake
    );
    ...
}
```

3S-IndexTokenStaking-M02

UpdatedStakerState Account can't be initialized DoSing **update_staker_state**

Id	3S-IndexTokenStaking-M02
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #728854a.

Description

When updating the **distribution_duration** by calling **update_staker_state** function, **UpdatedStakerState** should be provided as **info**, where it should be an already initialized account.

index_token_staking/src/lib.rs#L72-L73

```
pub struct UpdateStakerState<'info> {
    ...
    #[account(mut)]
>>  pub updated_staking_state: Account<'info, UpdatedStakerState>,
    pub token_program: Interface<'info, TokenInterface>,
}
```

The problem is that this account can't be initialized in any part in the program. There is no way for users to initialize that account and use it for updating their **staking_state.distribution_duration**.

The only way for Authors to do this is to deploy a custom program, and create an Account has the same struct, and use it. which is not the normal process especially for normal users who don't know how to deploy programs on Solana and do such a process.

Recommendations

make the account with **init_if_needed**, since Same **UpdatedStakerState** can be used by more than one Author, there is no need to create new one each time. And the seed can be a constant independent to any parameter.

3S-IndexTokenStaking-L01

Incorrect Event Emit at **DepositRewardTokenEvent**

Id	3S-IndexTokenStaking-L01
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #ab7bf55.

Description

When emitting the event when depositing rewards, the **from** parameter is put as **authority** instead of his **ATA** account.

index_token_staking/src/instructions/deposit_reward_token.rs#L133

```
pub fn handle_deposit_reward_token( ... ) -> Result<()> {
    ...
    let cpi_accounts = TransferChecked {
>>      from: ctx.accounts.owner_reward_token_account.to_account_info(),
        to: ctx.accounts.reward_token_vault_account.to_account_info(),
        authority: ctx.accounts.authority.to_account_info(),
        mint: ctx.accounts.reward_token_mint.to_account_info(),
    };
    ...
    emit!(DepositRewardTokenEvent {
>>      from: ctx.accounts.authority.key(),
        to: ctx.accounts.reward_token_vault_account.key(),
        ...
    });
    Ok(())
}
```

The Transferring process if from ATA to another ATA account, we can see that **from** is **owner_reward_token_account**, but in the event we put the signer **authority** key instead.

Recommendations

Change **authority** and put **owner_reward_token_account**

3S-IndexTokenStaking-L02

Unchecking **distribution_duration** value when updating

Id	3S-IndexTokenStaking-L02
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #b4bb89b.

Description

When creating (initializing) new staking account. we enforce distribution_duration to be greater than 5.

index_token_staking/src/instructions/initialize.rs#L57

```
pub fn handle_initialize(ctx: Context<Initialize>, distribution_duration: u64) -> Result<()> {
    ...
>>  require!(distribution_duration >= 5, CustomError::InvalidTimeRange);
    ...
}
```

But when updating this value by calling **update_staker_state** from the Authority, there is no check weather the new distribution_duration is actually greater than 5 or not.

index_token_staking/src/lib.rs#L32

```
    pub fn update_staker_state(ctx: Context<UpdateStakerState>) -> Result<()> {
        let staking_state = &mut ctx.accounts.staking_state;
        require!(ctx.accounts.owner.key() == staking_state.update_authority,
CustomError::Unauthorized);
>>      staking_state.distribution_duration =
ctx.accounts.updated_staking_state.distribution_duration;
        Ok(())
    }
```

Recommendations

Check that **updated_staking_state.distribution_duration** is greater than 5 to prevent **InvalidTimeRange** to be set.

3S-IndexTokenStaking-L03

Updating_distribution_duration is not checking the finalization of the prev Round

Id	3S-IndexTokenStaking-L03
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #ac6d45d.

Description

Updating_distribution_duration can be called if there is already a rewards round that will be distributed. This seems to be OK, as the new value will be used for the next rounds only, but in case it is fired before a given Round finish, then there is a reward a new reward distribution occur before the first one finish, the distribution of the first reward will be incorrectly handled by the new duration, instead of the original one he goes with.

index_token_staking/src/lib.rs#L27-L35

```
    pub fn update_staker_state(ctx: Context<UpdateStakerState>) -> Result<()> {
        let staking_state = &mut ctx.accounts.staking_state;
        require!(ctx.accounts.owner.key() == staking_state.update_authority,
CustomError::Unauthorized);
>>      staking_state.distribution_duration =
ctx.accounts.updated_staking_state.distribution_duration;
        Ok(())
    }
```

- If we distribute **100** tokens for **7** days duration

- Then, after **3.5 days** passed we changed the duration of the next rewards to **30** days, and distributed **400** token.

- The left **50** token from the first **100** will goes with distribution for **30** days instead of **7 days**

- This will affect the reward distributions durations, as this reward is just tied with **7 days** staking, elongating it to be **30** days is incorrect

- Same as shorted the duration, it will make the money to be staked for long period, goes at less intervals.

- This will make an infair process for distribution of the rewards, where the Stakers will not each the same reward for same tokens / same duration.

Recommendations

Prevent updating **Updating_distribution_duration** unless the last distribution already finished

3S-IndexTokenStaking-L04

new_reward_per_token is of type **u64** Can lead to Complete DoS of the protocol if it reached the **u64::MAX**

Id	3S-IndexTokenStaking-L04
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #3280350.

Description

The data type of **new_reward_per_token** is **u64**, this seems to be enough for occupying staking for long periods (years).

The value is always increasing, so we should guarantee that it will not reach the max at any circumstances.

index_token_staking/src/state.rs#L64-L68

```
    pub fn get_reward_per_token(&self, timestamp: u64) -> Result<u64> {
        ...
>>      let reward = (time_diff as u128)
            .checked_mul(self.reward_rate)
            .ok_or_else(|| error!(CustomError::MathError))?
            .checked_mul(
                10_u128
                    .checked_pow(self.base_decimals as u32)
                    .ok_or_else(|| error!(CustomError::MathError))?,
            )
            .ok_or_else(|| error!(CustomError::MathError))?
            .checked_div(self.total_staked as u128)
            .ok_or_else(|| error!(CustomError::MathError))?;
>>      let new_reward_per_token = (self.reward_per_token as u128)
            .checked_add(reward)
            .ok_or_else(|| error!(CustomError::MathError))?
            .try_into()
            .map_err(|_| error!(CustomError::MathError))?;
```

```
    Ok(new_reward_per_token)
}
```

It will always be increased with the value of the **reward**. the reward value is calculated as following equation:

- **time_diff * reward_rate * BASE / total_staked**

- **time_diff * reward_rate** will be the tokens to be distributed at this round if the global checkpoint (new_reward_per_token) only updated one time after the period finished.

- BASE is simple 1e<decimals> of stake token, we will do calculations with **9** decimals as most SPL tokens are **9** decimals.

- So the equation will be **reward * 1e9 / total_staked**, by going with the reward_token_mint as **9** decimals too, so undernormal situations, the value should be of **9** decimals, so it is safe to store at **u64**, which its max is ~= **18.45e18**

- The problem is that in case there is a heavy withdrawing from the stakers before the total reward is finalized, where only some weis left in **total_staked**, like **1-9**. So this will make the dividing by **total_staked** is like nothing, making the **new_reward_per_token** value goes increased by ~ **reward * 1e9**

- Since reward is of 9 decimals too, it is like you are adding of **18** powering, making you goes too near the **u64::MAX**

- If in some situations this case occurs and **total_staked** goes increased by HIGH amounts, and it reaches after more than one cycle and execution to **18.2e18**. Then it is a matter of time till the function goes reverted with overflow error. NOTE: the value is always increasing.

- We can't determine when it will reach the **u64::MAX**, but in case there was a lot of money in the Pool, and the staking is going normally (after the situations of few weis in **total_staked**, and this value reached the MAX, the protocol will go with incomplete DoS process, making all Funds Locked, as this function is executed whenever there is a deposit_reward/stake/unstake/claim_reward.

This will make the Funds completely frozen and unrecoverable.

Recommendations

Make the variable of type **u128** in **StakerState**, and **UserStakingAccount**

3S-IndexTokenStaking-L05

Missing remainder handling in reward-rate calculation allows permanent reward-token dust to accumulate in the vault

Id	3S-IndexTokenStaking-L05
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #3280350.

Description

IndexTokenStaking::handle_deposit_reward_token recalculates **reward_rate** with plain integer division, silently discarding the remainder every time new rewards are deposited. This occurs in both execution branches of the handle_deposit_reward_token function. As a result, **staking_state.total_reward** is incremented by the full deposit while the streaming mechanism can release only the truncated amount, leaving the leftover tokens (“dust”) locked in the contract and breaking accounting invariants.

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/instructions/deposit_reward_token.rs#L84C5-L104C6:

```
// branch A — previous period finished
staking_state.reward_rate = amount_u128
    .checked_div(distribution_duration_u128)?;
// branch B — previous period still running
let total_new_reward = amount_u128 + remaining_reward;
staking_state.reward_rate = total_new_reward
    .checked_div(distribution_duration_u128)?;
```

Steps to Reproduce

1. Set **distribution_duration** to 1 000 seconds.
2. After the prior period has ended, call **deposit_reward_token** with **amount = 1 001**.
3. Observe on-chain state: **reward_rate == 1**, **total_reward == 1 001**.

4. When the new period finishes only 1 000 tokens are streamed; the extra token remains in the vault indefinitely.

The truncation scales with **distribution_duration**. With a six-decimal reward token and a seven-day period, a single top-up can strand up to 604 800 base units (~0.6048 tokens). Repeated deposits compound the discrepancy, causing persistent accounting drift.

Recommendation

Make sure to multiply the deposited amount by the base scale before storing the new reward_rate and total_reward to avoid the loss of dust amounts.

```
- staking_state.reward_rate = amount_u128
-     .checked_div(distribution_duration_u128)?;
- staking_state.total_reward = amount_u128;
+ let base_multiplied_amount = amount_u128
+     .checked_mul(base);
+ staking_state.reward_rate = base_multiplied_amount
+     .checked_div(distribution_duration_u128)?;
+ staking_state.total_reward = base_multiplied_amount;
```

Ensure the necessary adaptations to the other methods that utilize reward_rate and total_reward are made to properly account for the base factor multiplication.

3S-IndexTokenStaking-L06

Early exit when **total_staked** drops to zero leads to permanently stranded reward tokens

Id	3S-IndexTokenStaking-L06
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #f91be07.

Description

StakerState::get_reward_per_token converts the global **reward_rate** into rewards per staked token. To avoid a divide-by-zero, it returns immediately whenever **total_staked == 0**. This short-circuit is safe during normal operation because **DepositRewardToken::handle_deposit_reward_token** already enforces **total_staked ≥ 10^base_decimals**, preventing the pool from being pre-funded before anyone stakes.

The problem arises later: if every user fully unstakes before the current reward period ends, **total_staked** becomes zero while tokens are still being streamed into **reward_token_vault_account**. During this interval **get_reward_per_token** keeps early-exiting, so the accumulated tokens are never reflected in **reward_per_token** and can never be claimed.

When a new user eventually stakes, **claim_rewards** starts a fresh accounting era, irrevocably discarding all rewards that accrued while TVL was zero. Those tokens remain locked in the vault, slightly understating APR and breaking the invariant that “all streamed rewards are eventually distributed”.

Steps to Reproduce

1. Start a reward period with at least one staker and call **DepositRewardToken::handle_deposit_reward_token** to fund it.

2. Let rewards stream for _t_ seconds where **0 < t < distribution_duration**.

3. All stakers unstake, driving **total_staked** to zero. Rewards continue streaming for **distribution_duration − t** seconds.

4. A new user stakes. Observe:

   * **reward_per_token** has not increased since step 2.

* **unclaimed_rewards** for the new staker is zero.

* **reward_token_vault_account** balance is larger by **reward_rate × (distribution_duration − t)**.

* The excess balance is now unreachable.

Recommendation

Buffer rewards accrued while **total_staked == 0** and inject them on the next call when stake exists:

```
+    pub pending_reward: u128, // new storage slot
     let accrued = (time_diff as u128) * self.reward_rate;
     if self.total_staked == 0 {
-        return Ok(self.reward_per_token);
+        self.pending_reward += accrued;          // hold until someone stakes
+        return Ok(self.reward_per_token);
     }
-    let distributable = accrued;
+    let distributable = accrued + self.pending_reward;
+    self.pending_reward = 0;
```

Alternatively, add an update_authority-only escape hatch (e.g., **sweep_unclaimed_rewards**) to allow rescuing undistributed tokens.

3S-IndexTokenStaking-N01

Imprecise Error Semantics

Id	3S-IndexTokenStaking-N01
Classification	None
Category	Suggestion
Status	Addressed in #080f2a5.

Description

Several arithmetic operations use **checked_sub**, which can only fail by **underflow**, yet the code maps that failure to unrelated variants such as **StakeOverflow**, **TimeOverflow**. Examples:

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/83853614c88f59185a2566bd6d49d23c76a4ce86/programs/index_token_staking/src/instructions/unstake.rs#L64

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/83853614c88f59185a2566bd6d49d23c76a4ce86/programs/index_token_staking/src/instructions/stake.rs#L91

The catch-all **MathError** is also used then there are defined errors for the custom case. Examples:

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/83853614c88f59185a2566bd6d49d23c76a4ce86/programs/index_token_staking/src/state.rs#L50

https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/83853614c88f59185a2566bd6d49d23c76a4ce86/programs/index_token_staking/src/instructions/deposit_reward_token.rs#L92

Recommendation

Use precise variants:

StakeUnderflow, **TimeUnderflow**, etc., for **checked_sub** failures.

Reserve ***Overflow** for **checked_add**/**checked_mul** failures.

Keep **MathError** only for mixed-op branches where both directions are plausible.

3S-IndexTokenStaking-N02

Dead code artifacts (unused events, constants, structs, and error codes) allow maintenance mistakes

Id	3S-IndexTokenStaking-N02
Classification	None
Category	Suggestion
Status	Addressed in #3e1abf4.

Description

The current codebase defines several items that are never referenced elsewhere:

The **ClaimRewards** event at https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/events.rs

The **UNSTAKING_PERIOD** constant at https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/constants.rs

The **MintAuthority** account struct at https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/state.rs

Six **CustomError** variants (**InvalidDepositAmount**, **StakingNotStarted**, **UnstakingHasNotEnded**, **AlreadyStaking**, **MustBeUnstaking**, **InsufficientRewards**) at https://github.com/OpenDeltaLabs/index_token_staking_audit/blob/main/programs/index_token_staking/src/errors.rs

Recommendation

Delete the unused definitions or implement the missing logic if they are part of planned functionality. This keeps the codebase minimal, ensures every error path is reachable, and prevents silent gaps in validation.

3S-IndexTokenStaking-N03

Protocol is incompatible with **fee-on-transfer** tokens

Id	3S-IndexTokenStaking-N03
Classification	None
Category	Suggestion
Status	Addressed in #ee10dd4.

Description

The Staking Program deals that the **amount** of tokens to be transferred by **from** will be the same as that received by **to**.

Since the Protocol uses **2022** SPL tokens, they come with extensions, and form this extensions are **fee on transfer** value.

Fees will be taken from the amount, so the Vault will not store the exact amount it receive, leading to out of funds issue when unstaking all of the tokens

Recommendations

When staking, don't store the amount, instead store the exact amount received by the user.

3S-IndexTokenStaking-N04

Missing Checking Insufficient Rewards in withdrawing

Id	3S-IndexTokenStaking-N04
Classification	None
Category	Suggestion
Status	Addressed in #bc80a79.

Description

When withdrawing the unclaimed rewards there is no check weather the value to be withdrawn is zero or not.

index_token_staking/src/instructions/withdraw_rewards.rs#L90

```
pub fn handle_withdraw_rewards(ctx: Context<WithdrawRewards>) -> Result<()> {
    ...
>>  let total_withdrawable_rewards = user_staking_account.unclaimed_rewards;
    user_staking_account.unclaimed_rewards = 0;
    msg!("Reward: {}", total_withdrawable_rewards);
    ...
}
```

There is no check for the value of **unclaimed_rewards**, if it is greater than **0** or not. Where there is an error message **InsufficientRewards**, which should be used if there is no reward to withdraw, but it is not used. Leading to fire a withdrawal transaction by the Vault even with 0 amount rewards.

Recommendations

Check that the **unclaimed_rewards** is greater than **0** and revert with **InsufficientRewards** error, if it is zero.

3S-IndexTokenStaking-N05

last_updated_timestamp is updated twice

Id	3S-IndexTokenStaking-N05
Classification	None
Category	Suggestion
Status	Addressed in #1f15fb0.

Description

When calling **handle_deposit_reward_token** the value of **staking_state.last_updated_timestamp** is updated two times. at the beginning and the end of the function execution.

index_token_staking/src/instructions/deposit_reward_token.rs#L115

```
pub fn handle_deposit_reward_token(ctx: Context<DepositRewardToken>, amount: u64) ->
Result<()> {
    ...
>>  staking_state.last_updated_timestamp = now;
    ...
    require!(max_reward_scaled <= u128::MAX, CustomError::RewardRateTooHigh);
>>  staking_state.last_updated_timestamp = now;
    ...
}
```

Since it is already updated in the beginning there is no need to update one more time.

Recommendations

Remove the second update of this variable