CERTIK

# TOTT STAKING - Audit

## Security Assessment

CertiK Assessed on Apr 2nd, 2026


CertiK Assessed on Apr 2nd, 2026

# TOTT STAKING - Audit

The security assessment was prepared by CertiK.

## Executive Summary

| TYPES | ECOSYSTEM | METHODS |
|---|---|---|
| Staking | Solana (SOL) | Manual Review, Static Analysis |

| LANGUAGE | TIMELINE |
|---|---|
| Rust | Preliminary comments published on 02/18/2026 |
| | Final report published on 04/02/2026 |

## Vulnerability Summary

| 9 | 9 | 0 | 0 | 0 |
|---|---|---|---|---|
| Total Findings | Resolved | Partially Resolved | Acknowledged | Declined |

| | | | |
|---|---|---|---|
| 0 | Centralization | | Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets. |
| 0 | Critical | | Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks. |
| 2 | Major | 2 Resolved | Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control. |
| 2 | Medium | 2 Resolved | Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform. |
| 2 | Minor | 2 Resolved | Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions. |
| 3 | Informational | 3 Resolved | Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code. |

# TABLE OF CONTENTS | TOTT STAKING - AUDIT

# AUDIT SCOPE | TOTT STAKING - AUDIT

you6878/smart_contract_tott


lib.rs

# APPROACH & METHODS | TOTT STAKING - AUDIT

This audit was conducted for TOTT to evaluate the security and correctness of the smart contracts associated with the TOTT STAKING - Audit project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

# REVIEW NOTES | TOTT STAKING - AUDIT

## Overview

This project contains a single Solana Anchor program ( `spl_staking` ) implementing an SPL-token staking pool with fixed lock options (6/9/12 “months” using a 30/360 convention), periodic reward claiming, and early-unstake penalties.

The system uses a program-derived vault token account ( `token_vault` ) as custody for deposits and as the payout source for rewards and withdrawals.

1. `lib.rs`

The `spl_staking` program manages a staking pool ( `StakingPool` ) and per-user staking positions ( `UserStake` ) keyed by a user-supplied `index` .

Key characteristics:

- **Pool initialization**
  - `initialize_pool()` creates a canonical `staking_pool` PDA derived from `[b"staking_pool", token_mint]` .
  - Initializes a `token_vault` SPL token account derived from `[b"token_vault", token_mint]` with authority = staking_pool PDA.
  - Stores an `authority` pubkey in state, but there are no admin-gated instructions in this version (the field is currently unused for authorization).
- **Position model**
  - Each stake position is a `UserStake` PDA derived from:
    - `[b"user_stake_v4", user, staking_pool, index]`
  - `stake()` transfers user tokens into `token_vault` and records amount/timestamps/lock period and an effective “bonus rate”.
- **Reward accounting**
  - Uses integer month rounding with `SECONDS_PER_MONTH = 30 days` .
  - `claim_rewards()` can be called when at least one full month has elapsed since the last claim.
  - Rewards are paid from the same `token_vault` via CPI transfer signed by the `staking_pool` PDA.
- **Unstaking**
  - `unstake()` supports:
    - **Early unstake**: returns ((principal - 90% penalty) + unclaimed interest)
    - **Mature unstake**: returns (principal + unclaimed interest)

- Payouts are made from `token_vault` using the `staking_pool` PDA as the token authority.

Operational/economic correctness therefore depends on:

- **Vault solvency**: rewards and withdrawals are paid from `token_vault`, so liquidity management/top-ups directly affect ability to pay rewards and, in underfunded scenarios, even withdrawals.
- **Index management**: positions are keyed by a user-supplied `index`; integrators must track indices reliably (no events are emitted in this version).

---

## External Dependencies

| Dependency | Notes |
|---|---|
| Solana runtime | `Clock` sysvar timestamps used for accrual/lock logic |
| Anchor (`anchor_lang`) | Account validation, PDA derivation, program framework |
| Anchor SPL (`anchor_spl::token`) | CPI to SPL Token for transfers |
| SPL Token Program | Token custody and transfer enforcement |

## Privileged Functions / Trust Assumptions

### Pool / Initialization

- **Any signer (permissionless)**
  - `initialize_pool()`: first initializer for a given mint sets up the canonical pool/vault (and writes `pool.authority`, though it is not used for access control in this version).

### Users

- **Users (signers)**
  - `stake(amount, lock_period_months, index)`
  - `claim_rewards(index)`
  - `unstake(amount, is_early, index)`

### Notes on centralization / admin control

There are **no admin withdrawal or parameter-change functions** in this code. However:

- Pool creation is first-writer-wins per mint.
- Ongoing safety and UX depend on operational funding/solvency of the vault and on reliable off-chain indexing/position tracking.

# FINDINGS | TOTT STAKING - AUDIT

| 9 | 0 | 0 | 2 | 2 | 2 | 3 |
|---|---|---|---|---|---|---|
| Total Findings | Critical | Centralization | Major | Medium | Minor | Informational |

This report has been prepared for TOTT to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 9 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

| ID | Title | Category | Severity | Status |
|---|---|---|---|---|
| TSA-02 | Vault Insolvency Risk From Rewards Paid Out Of Principal Vault | Design Issue | Major | Resolved |
| TSA-03 | Early Unstake Penalty Can Be Reduced Via Rounding And Partial Unstake Splitting | Incorrect Calculation | Major | Resolved |
| TSA-01 | Permissionless Pool Initialization Allows Arbitrary Authority Assignment (Latent Governance Risk) | Governance | Medium | Resolved |
| TSA-04 | Missing UserStake Account Closure Results In Permanent Rent Loss | Design Issue | Medium | Resolved |
| TSA-05 | Zero-Amount Stake Allowed | Logical Issue | Minor | Resolved |
| TSA-06 | Saturating Arithmetic Can Mask Accounting Inconsistencies | Logical Issue | Minor | Resolved |
| TSA-07 | Missing PDA Seed Validation For Staking_pool In Stake Context | Access Control | Informational | Resolved |
| TSA-08 | Missing Event Emission Reduces Observability And Position Discoverability | Design Issue | Informational | Resolved |
| TSA-09 | Missing Explicit Has_one = User Constraint Reduces Defense-In-Depth | Access Control | Informational | Resolved |

# TSA-02 | Vault Insolvency Risk From Rewards Paid Out Of Principal Vault

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Design Issue | Major | lib.rs: 274~299, 337~342 | Resolved |

## Description

The staking program uses a single token vault ( `token_vault` ) to custody user principal while also serving as the source of reward payouts and unstake redemptions. Rewards are not minted and there is no protocol-native funding mechanism to guarantee reward liquidity. As a result, all reward claims directly reduce the same vault balance that must later satisfy principal withdrawals.

In `claim_rewards()` , rewards are transferred directly from `token_vault` after a balance check:

```
require!(ctx.accounts.token_vault.amount >= required,
StakingError::InsufficientVaultBalance);
```

Similarly, `unstake()` enforces that the vault must be able to cover the total owed amount:

```
require!(
    ctx.accounts.token_vault.amount >= total_to_user,
    StakingError::InsufficientVaultBalance
);
```

Because both reward payouts and principal withdrawals depend on the same liquidity pool, the system implicitly assumes that sufficient external funding will always exist to cover reward obligations. However, no on-chain mechanism enforces or guarantees this.

This creates a structural insolvency condition:

- Rewards paid to early claimers reduce vault liquidity.
- If cumulative reward payouts exceed externally supplied surplus tokens,
- The vault balance may fall below what is required to satisfy later unstake() calls.
- Subsequent unstake attempts revert, even when the user is only attempting to withdraw their principal.

In this state, user funds are not technically stolen, but they become operationally locked until additional tokens are manually transferred into the vault. Since `REWARD_RESERVE_AMOUNT` is configured as `0` , the reserve guard provides no effective solvency buffer.

The economic model therefore exhibits a first-come-first-served liquidity dependency, where early reward claimers are advantaged and later users bear availability risk. This undermines the assumption that staked principal is always withdrawable.

## Recommendation

Enforce strict Principal Protection so user deposits are never used to pay rewards.

**Recommended**: Segregate funds by using two vaults: a Principal Vault that only holds user deposits (never used for rewards) and a separately funded Reward Vault used exclusively for interest payouts. If the Reward Vault is depleted, reward claims fail, but principal withdrawals remain fully available.

**Alternative (If a single-vault model is retained)**: `Gate claim_rewards()` by paying rewards only from excess liquidity, i.e., allow rewards only when `token_vault.amount - principal_liability >= reward_amount` (where `principal_liability` is the total principal owed to users). This guarantees the vault cannot be drained below the amount required to return principal.

## Alleviation

**[TOTT, 03/25/2026]**: The team partially mitigated the issue by restricting `claim_rewards()` to surplus vault balance above total staked principal; however, matured `unstake()` still distributes rewards from the same vault without ensuring global principal coverage, verified in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

---

**[TOTT, 04/02/2026]**: I added Alternative (If a single-vault model is retained): `Gate claim_rewards()` by paying rewards only from excess liquidity, i.e., allow rewards only when `token_vault.amount - principal_liability >= reward_amount` (where `principal_liability` is the total principal owed to users). This guarantees the vault cannot be drained below the amount required to return principal.

into unstake

---

**[TOTT, 04/02/2026]**: The team heeded the advice and resolved the issue by enforcing surplus-only reward payouts in both claim_rewards() and unstake(), ensuring reward distributions cannot reduce principal coverage, in commit ef63851ce58a2de5ed75b027066ceead87237205

# TSA-03 Early Unstake Penalty Can Be Reduced Via Rounding And Partial Unstake Splitting

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Incorrect Calculation | Major | lib.rs: 125~135 | Resolved |

## Description

The early unstake penalty is computed using integer floor division:

```
let penalty_u128 = (amount as u128)
    .checked_mul(EARLY_UNSTAKE_PENALTY_RATE as u128)?
    .checked_div(100u128)?;
```

This implements `penalty = floor(amount × 90 / 100)`, which rounds down. As a concrete example, for `amount = 1`, the computed penalty becomes `0`. Since `unstake()` supports repeated partial early withdrawals, a user can split an early exit into many small unstakes and pay a lower aggregate penalty than intended compared to withdrawing the same total amount in a single transaction. The practical magnitude depends on token decimals (i.e., base-unit granularity), but the logic breaks the protocol's intended invariant that early exit should consistently incur the configured 90% penalty.

## Scenario

1. A user stakes tokens and intends to exit early (should incur 90% penalty).
2. Instead of a single early withdrawal, the user performs many early partial unstakes with small `amount` values.
3. Due to truncation, some partial withdrawals incur reduced penalties (e.g., `amount = 1 → penalty = 0`).
4. Over multiple transactions, the total penalty paid is reduced compared to a single early withdrawal of the same total amount.

## Recommendation

Use ceiling division when computing the penalty to eliminate truncation advantage and ensure the effective penalty cannot fall below the intended rate.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by applying ceiling division in penalty calculation, preventing rounding-based reduction through partial unstake splitting, in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-01 Permissionless Pool Initialization Allows Arbitrary Authority Assignment (Latent Governance Risk)

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Governance | Medium | lib.rs: 199~207, 360~388 | Resolved |

## Description

The `initialize_pool()` instruction is fully permissionless. Any signer may initialize the canonical `staking_pool` PDA for a given `token_mint` and arbitrarily set the `pool.authority` field:

```
pub fn initialize_pool(ctx: Context<InitializePool>) -> Result<()> {
    let pool = &mut ctx.accounts.staking_pool;
    pool.authority = ctx.accounts.authority.key();
    pool.token_mint = ctx.accounts.token_mint.key();
    pool.token_vault = ctx.accounts.token_vault.key();
    pool.total_staked = 0;
    pool.bump = ctx.bumps.staking_pool;
    Ok(())
}
```

The corresponding `InitializePool` account struct allows any signer to act as `authority`:

```
#[account(mut)]
pub authority: Signer<'info>,
```

Because the staking pool PDA is derived deterministically from the token mint:

```
seeds = [b"staking_pool", token_mint.key().as_ref()],
```

The first initializer permanently determines the stored authority for that mint (first-writer-wins).

Although `pool.authority` is currently unused for authorization in any instruction, this creates a latent governance/control-plane risk. Any future program upgrade, administrative feature, or off-chain integration that assumes `pool.authority` represents the legitimate admin could inadvertently grant privileged status to an unintended party.

The presence of an unused admin field combined with permissionless initialization introduces future operational and governance risk.

## Recommendation

- If the protocol is intended to be fully permissionless and immutable, remove the authority field entirely from `StakingPool` to avoid misleading governance assumptions.

- If administrative control is intended, restrict `initialize_pool()` so only a predefined deployer/admin can initialize the pool, and ensure all future privileged actions explicitly validate `pool.authority`.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by restricting `initialize_pool()` to a predefined admin, preventing arbitrary authority assignment, in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-04 Missing UserStake Account Closure Results In Permanent Rent Loss

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Design Issue | Medium | lib.rs: 228~241 | Resolved |

## Description

Each staking position is represented by a dedicated `UserStake` PDA created with `init`, meaning the user funds and locks rent-exempt SOL when the account is created. However, when a user fully unstakes, the program merely resets the account fields instead of closing the account and returning the locked lamports.

```
if user_stake.amount == 0 {
    user_stake.stake_timestamp = 0;
    user_stake.lock_period_months = 0;
    user_stake.bonus_rate = 0;
    user_stake.last_claim_timestamp = 0;
    user_stake.user = Pubkey::default();
}
```

While the stake data is cleared, the account itself remains allocated on-chain. Since there is no `close = user` constraint or dedicated close instruction, the rent-exempt SOL used to create the account becomes permanently unrecoverable.

This issue has two important consequences:

- **Permanent user-level capital loss:** Each staking position results in an irreversible rent cost once fully withdrawn.
- **Compounding impact due to index-based design:** Because the program supports multiple positions per user (via index), repeated staking cycles lead to cumulative SOL loss.
- **On-chain state bloat:** “Zombie” `UserStake` accounts persist indefinitely, increasing storage footprint without functional purpose.

Although this does not directly compromise token balances, it results in avoidable economic loss to users and inefficient long-term state management.

## Recommendation

- Remove the field-reset logic on full unstake and reclaim rent by closing the account instead. Implement a dedicated `close_stake` instruction that requires `UserStake.amount == 0` and uses Anchor’s `close = user` to refund rent-exempt lamports to the owner.
- Alternatively, add an `unstake_all` instruction (with a separate accounts context) that performs a full withdrawal and closes the `UserStake` account atomically, ensuring partial unstakes can never trigger an unintended close.

## Alleviation

**[TOTT, 03/31/2026]**: The team heeded the advice and resolved the issue by ensuring the user field is no longer reset on full unstake, allowing the `has_one = user` constraint to pass and enabling proper account closure and rent recovery, in commit 8502e3e88ebb439738aafd387554603123500e9b

# TSA-05 | Zero-Amount Stake Allowed

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Logical Issue | Minor | lib.rs: 212~213 | Resolved |

## Description

The `stake()` function does not validate that the provided `amount` is greater than zero:

```
pub fn stake(ctx: Context<Stake>, amount: u64, lock_period_months: u64, _index: u64)
-> Result<()> {
```

There is no `require!(amount > 0, ...)` check prior to executing the token transfer. Since SPL token transfers allow transferring `0` tokens, a user can call:

```
stake(amount = 0, ...)
```

This will:

- Successfully execute the instruction.
- Initialize a new `UserStake` PDA.
- Lock rent-exempt SOL from the user.
- Set stake state values with `amount = 0`.

The resulting position is effectively unusable:

- It cannot meaningfully accrue rewards.
- It cannot produce meaningful withdrawals.
- It permanently consumes rent unless manually closed (which is currently not implemented).

This introduces unnecessary state bloat and creates avoidable economic loss for users who may accidentally or maliciously create zero-value stake positions.

## Recommendation

Add an explicit validation at the beginning of `stake()`

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by enforcing a non-zero amount check in `stake()` in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-06 | Saturating Arithmetic Can Mask Accounting Inconsistencies

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Logical Issue | Minor | lib.rs: 177~193 | Resolved |

## Description

After an unstake, the program updates both the user position and global pool accounting using saturating subtraction:

```
fn update_stake_after_unstake(
    user_stake: &mut UserStake,
    staking_pool: &mut StakingPool,
    amount: u64,
) {
    user_stake.amount = user_stake.amount.saturating_sub(amount);
    staking_pool.total_staked = staking_pool.total_staked.saturating_sub(amount);

    if user_stake.amount == 0 {
        user_stake.stake_timestamp = 0;
        user_stake.lock_period_months = 0;
        user_stake.bonus_rate = 0;
        user_stake.last_claim_timestamp = 0;
        user_stake.user = Pubkey::default();
    }
}
```

While saturating arithmetic prevents underflow panics, it can also silently clamp values to zero if an unexpected state mismatch ever occurs (e.g., due to integration mistakes, future code changes, or corrupted/incorrect assumptions). This reduces the protocol's ability to fail fast and can mask accounting drift between `staking_pool.total_staked` and the true sum of user positions

This is not an immediate exploit in the current flow (since `unstake()` checks `user_stake.amount >= unstake_amount` before calling this function), but it is a robustness concern that weakens invariant enforcement.

## Recommendation

Prefer strict invariant enforcement for accounting updates (i.e., revert on impossible underflow conditions) rather than saturating arithmetic, so state inconsistencies are detected early and cannot be silently masked.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by replacing saturating arithmetic with checked operations that revert on underflow, ensuring accounting inconsistencies cannot be silently masked, in commit

a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-07 | Missing PDA Seed Validation For Staking_pool In Stake Context

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Access Control | Informational | lib.rs: 402~404 | Resolved |

## Description

In the `Stake` instruction context, the `staking_pool` account is declared as:

```
#[account(mut)]
pub staking_pool: Account<'info, StakingPool>,
```

Unlike the `UnstakeCtx` and `ClaimRewards` contexts, which enforce canonical PDA derivation using:

```
seeds = [b"staking_pool", staking_pool.token_mint.as_ref()],
bump = staking_pool.bump
```

Although indirect constraints exist (e.g., token mint matching and vault address validation), the absence of explicit seed validation weakens defense-in-depth and creates inconsistency across instruction flows.

While no direct exploit is currently evident due to the additional constraints in place, failing to enforce canonical PDA derivation increases future upgrade risk and may introduce unintended state assumptions if validation logic is modified.

## Recommendation

Add explicit PDA seed and bump validation for `staking_pool` in the `Stake` context, matching the constraints already enforced in `UnstakeCtx` and `ClaimRewards`.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by enforcing PDA seed and bump validation for staking_pool in the Stake context, ensuring consistent canonical account verification, in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-08 Missing Event Emission Reduces Observability And Position Discoverability

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Design Issue | Informational | lib.rs: 390~420 | Resolved |

## Description

The program does not emit structured Anchor events (#[event] + emit!()) for key protocol state transitions, including:

- initialize_pool()
- stake()
- claim_rewards()
- unstake()

Stake positions are derived using a user-supplied index in PDA seeds:

```
#[derive(Accounts)]
#[instruction(amount: u64, lock_period_months: u64, index: u64)]
pub struct Stake<'info> {
    #[account(
        init,
        payer = user,
        space = 8 + UserStake::LEN,
        seeds = [b"user_stake_v4", user.key().as_ref(), staking_pool.key().as_ref(),
&index.to_le_bytes()],
        bump
    )]
    pub user_stake: Account<'info, UserStake>,
    // ...
}
```

Because no events are emitted when a position is created, claimed, or withdrawn, off-chain systems must rely on instruction decoding and/or program-account indexing (e.g., scanning UserStake accounts) to track user positions and protocol activity. This weakens observability and increases operational complexity for integrators and operators.

## Recommendation

Emit high-level events for `initialize_pool`, `stake`, `claim_rewards`, and `unstake` (including `user`, `index`, `amount`, and `timestamps`) to enable reliable off-chain position tracking and monitoring without heavy account scanning.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by adding structured event emissions for key state transitions, improving observability and off-chain tracking, in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# TSA-09 Missing Explicit Has_one = User Constraint Reduces Defense-In-Depth

| Category | Severity | Location | Status |
| --- | --- | --- | --- |
| Access Control | Informational | lib.rs: 231, 425~431, 458~463 | Resolved |

## Description

The `UserStake` account stores an ownership field:

```
pub user: Pubkey,
```

This field is set during staking:

```
user_stake.user = ctx.accounts.user.key();
```

However, in both `UnstakeCtx` and `ClaimRewards`, the stored `user` field is never explicitly validated against the transaction signer using an Anchor `has_one = user` constraint.

Ownership is currently enforced implicitly via PDA seed derivation:

```
seeds = [
    b"user_stake_v4",
    user.key().as_ref(),
    staking_pool.key().as_ref(),
    &index.to_le_bytes()
]
```

Since the PDA includes `user.key()` in its derivation, a different user cannot derive the same account, and access control is effectively enforced.

However:

- The stored user field is written but never validated.
- The program relies on a single enforcement layer (PDA seeds) instead of two independent checks.
- If PDA seeds were ever modified in a future upgrade or migration, the implicit ownership guarantee could be weakened without fallback validation.

This makes the stored user field effectively dead state and reduces defense-in-depth clarity.

## Recommendation

Add an explicit `has_one = user` constraint to `user_stake` in both `UnstakeCtx` and `ClaimRewards` to enforce `user_stake.user == user.key()` as a defense-in-depth measure, or remove the user field from `UserStake` if it is not intended to be validated.

## Alleviation

**[TOTT, 03/25/2026]**: The team heeded the advice and resolved the issue by adding explicit has_one = user constraints to validate ownership, strengthening access control with defense-in-depth, in commit a40d4b8ec9f9d3839e2147b5b6bda2551c4aa0c8

# APPENDIX | TOTT STAKING - AUDIT

## Finding Categories

| Categories | Description |
| --- | --- |
| Design Issue | Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories. |
| Incorrect Calculation | Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended. |
| Governance | Governance findings indicate issues related to the management of the code. |
| Logical Issue | Logical Issue findings indicate general implementation issues related to the program logic. |
| Access Control | Access Control findings are about security vulnerabilities that make protected assets unsafe. |

# DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

# Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.