# Staking-Pool Protocol Audit Report

0xOwain

September 9, 2025

# Staking-Pool Protocol Audit Report

Version 1.0

*Cyfrin.io*

September 3, 2025

# Staking-Pool Protocol Audit Report

0xOwain

September 9, 2025

Prepared by: 0xOwain Lead Auditors: - 0xOwain

## Table of Contents



## Protocol Summary

This protocol implements a simple staking pool for ERC20 tokens, allowing users to deposit ("stake") tokens, earn proportional rewards from injected fees anmd donations, and withdraw ("unstake") their tokens to a configurable fee.

Staking: - Users call `stake(uint256 amount)` to transfer tokens into the pool. A staking fee (0-10%, configurable by the owner) may be deducted and redistributed anmong existing stakers. The user's staked balance is updated, and their divident checkpoint (`round`) is recorded for reward calculations.

Unstaking: - Users can withdraw tokens via `unstake(uint256 amount)`. An unstaking fee (0-10%) may be deducted and redistributed across remaining stakers. Pending rewards ar automatically claimed and added to the withdrawn amount. The contract ensures the user cannot unstake more than they have staked.

Reward Distribution: - Rewards originate from two sources: 1. Collected staking/unstaking fees, which are redistributed proportionally among active stakers. 2. External injections via `donateToPool(uint256 amount)`, where any user can add ERC20 tokens to the reward pool. Rewards are tracked using a cumulative divident-per-token system with a scaling f acto to minimise rouning errors. Each staker's pending reward is calculated based on their stake and the difference in cumulative dividends since their last checkpoint.

Administration & Safety: - The owner (deployer) can pause/unpause the protocol, and adjust staking/unstaking fees (capped at 10% when set via `setFees`). - Emergency pause halts new staking/donations but still allows withdrawals. - The contract leverages OpenZeppelin's `Ownable` and `Pausable` modules.

Token Assumptions: - The pool assumes the staking token behaves as a standard ERC20 (18 decimals, retuns `true` on transfer, no rebasing or fee-on-transfer logic). Non-standard tokens may break accounting or reward distribution.

# Disclaimer

The 0xOwain team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

# Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

# Audit Details

## Scope

**Repo** https://github.com/mnedelchev-vn/staking-pool/tree/master

**Commit Hash** de4f2d9e2b1433b0a2c46f219866d109ed237ac2 (retrieved 30 Aug 2025)

**Contracts in Scope:** - StakingPool.sol: - Core staking logic with reward distribution and fees - 179 lines - TestERC20.sol: - Basic ERC20 mintable token for testing/staking - 13 lines

**Out of Scope:** - Counter.sol: - Unused Forge boilerplate - 14 lines - External dependencies

**Lines of Code (cloc) Summary**

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Solidity                         2             36             28            128
-------------------------------------------------------------------------------
SUM:                             2             36             28            128
-------------------------------------------------------------------------------
```

**Severity / Impact Criteria** - **High:** Loss/theft of funds, invariant violations enabling withdrawal > entitlement. - **Medium:** System insolvency over time (accounting drift), reward misallocation, privilege escalation. - **Low/Info:** Misleading events, missing input validation, precision/decimals caveats, docs.

## Roles

Owner: - Defined via OpenZeppelin's `Ownable`. - Privileges: - Pause/unpause the contract (`pause`, `unpause`). - Adjust staking and unstaking fees (`setFees`). - Limitations: - Cannot directly seize user funds. - Cannot bypass accounting to withdraw tokens other than via normal staking/unstaking logic.

Staker (User): - Any address that interacts with the pool. - Capabilities: - Stake tokens (`stake`). - Unstake tokens (`unstake`). - Claim pending rewards (`claimRewards`). - Contribute additional tokens as donations (`donateToPool`). - Responsibilities: - Must approve token transfers before staking. - Subject to staking/unstaking fees set by the Owner.

## Tools Used

- **Manual Review**: Full line-by-line analysis of `StakingPool.sol`, all findings confirmed through source code inspection
- **Slither v0.10.0**: Used for static analysis — flagged CEI violations, uninitialized variables, and pragma inconsistencies
- **Foundry** (`forge test`): Used to build custom fuzz tests and proof-of-concept exploits for reentrancy and logic flaws

- **Aderyn** : Static tool for additional pattern-based checks (e.g., missing `nonReentrant`, unsafe transfer patterns)

# Executive Summary

This report presents the findings from a security review of the `StakingPool` smart contract system. The protocol allows users to stake ERC-20 tokens, earn proportional rewards from donations, and withdraw their stake subject to optional fees. It includes configurable fee parameters, pausability, and a simple mechanism for distributing staking rewards over discrete rounds.

The audit uncovered 14 issues in total, including: - 2 High severity issues – related to reentrancy vulnerabilities and unsafe token compatibility assumptions - 3 Medium severity issues – covering constructor misconfigurations, accounting inaccuracies, and edge-case logic flaws - 9 Informational findings – covering naming, style, event transparency, and general best practices

The most critical issues stem from external token calls (e.g., `transfer`, `transferFrom`) made before internal state updates, violating the Checks-Effects-Interactions (CEI) pattern. These flaws were demonstrated via concrete proof-of-concept (PoC) exploits using malicious or fee-on-transfer tokens to trigger reentrancy or break internal assumptions.

The constructor also allows unsafe initial deployments (e.g., zero-address tokens, over-limit fees) that bypass the same checks enforced during regular operation. Additionally, the contract lacks defensive measures against non-standard ERC-20 behavior, which could lead to reward miscalculations or unfair pool behavior.

This audit did not assess frontend integration, off-chain systems, or economic/game-theoretic modeling. It assumes trusted ownership and standard deployment environments.

Despite these issues, the contract is structurally simple and generally well-written. With the recommended mitigations — including reentrancy protection, input validation, and safer ERC-20 handling — the `StakingPool` system can offer a reliable and robust staking mechanism.

## Issues found

| Severity | Number of Issues Found |
|---|---|
| High | 2 |
| Medium | 3 |
| Low | 0 |
| Informational | 9 |
| **Total** | **14** |

# Findings

## High

### [H-1] Reentrancy Risk via External Token Calls (`transfer` / `transferFrom`)

**Description** Critical functions make external calls to ERC20 tokens (`transfer`, `transferFrom`) before state updates, without any `nonReentrant` protection or pull-based design.

Affected functions: - `stake()` – `token.transferFrom(msg.sender, address(this), _amount)` - `unstake()` – `token.transfer(msg.sender, unstakingAmount)` - `donateToPool()` – `token.transferFrom(msg.sender, address(this), _amount)` - `claimReward()` (within `stake()` and `unstake()`) – `token.transfer(msg.sender, pendingReward)`

These calls assume standard ERC20 behavior but may behave unpredictably or trigger fallback hooks in ERC777 or malicious contracts. If an attacker reenters during these calls (e.g., before internal state is updated), it can lead to: - reward manipulation, - draining of tokens, - or denial-of-service (DoS) scenarios.

**Impact** High risk of reentrancy if non-standard/malicious tokens are used. Can lead to reward inflation, skipped accounting updates, or full compromise of the staking pool.

**Proof of Concept** PoC A – Reentrancy via `stake()` Transfer Call: This PoC simulates an ERC20 token that exploits the `transferFrom()` call during staking. The malicious token's `transferFrom()` triggers a reentrant call back into the staking pool (e.g., unstake(1)) before internal state is updated, exploiting the fact that state changes occur after the external call — violating the Checks-Effects-Interactions pattern. This allows reentrancy attacks during staking. The mock token below simulates a reentrant ERC20 that exploits transfer-based callbacks:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.28;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

interface IStakingPool {
    function unstake(uint256 _amount) external;
}

contract ReentrantERC20 is ERC20 {
    IStakingPool public stakingPool;
    bool public attackTriggered;

    constructor() ERC20("ReentrantToken", "REENT") {}
```

```
    function setTarget(address _stakingPool) external {
        stakingPool = IStakingPool(_stakingPool);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function _update(address from, address to, uint256 value) internal override {
        super._update(from, to, value);

        if (!attackTriggered && address(stakingPool) != address(0) && to == address(stakingP
            attackTriggered = true;
            stakingPool.unstake(1);
        }
    }

}
```

PoC B – Reentrancy via `claimReward()` Transfer Call: This PoC simulates an ERC20 token that triggers reentrancy during the `transfer()` call in `claimReward()`. The malicious token's internal transfer logic reenters into the pool and calls `unstake(1)`. Because internal accounting hasn't updated yet, this can lead to double-claiming rewards, early exits, or partial fund drains, especially if combined with token hooks (e.g., ERC777). This test validates that the reward claim logic violates CEI and can trigger internal reentrancy:

```
function test_Reentrancy_ClaimReward_TriggersUnstake() public {
    ReentrantERC20 reentrantToken = new ReentrantERC20();
    pool = new StakingPool(address(reentrantToken), 5, 5);
    reentrantToken.setTarget(address(pool));

    address attacker = address(1);
    vm.startPrank(attacker);

    reentrantToken.mint(attacker, 100);
    reentrantToken.approve(address(pool), 100);
    pool.stake(100);
    vm.stopPrank();

    reentrantToken.mint(address(this), 100);
    reentrantToken.approve(address(pool), 100);

    pool.donateToPool(100);

    vm.startPrank(attacker);
```

```
    vm.expectRevert();
    pool.claimReward();
    vm.stopPrank();
}
```

Note: This reentrancy attempt currently fails due to a guard condition (`InvalidAmount`), but it confirms a CEI violation. If internal validation logic changes or the attacker controls more state (e.g., pending rewards), the vector could become exploitable.

**Recommended Mitigation** - Add `nonReentrant` modifier to `stake()`, `unstake()`, and `donateToPool()`. - Refactor to follow checks-effects-interactions (CEI) pattern strictly. - Consider using pull-based reward claiming rather than automatic transfers. - Use SafeERC20 to catch non-standard token behavior.

## Medium

### [M-1] Constructor Input Validation and Initialization Issues

**Description**
The `StakingPool` constructor accepts critical parameters without full validation or consistent initialization practices:

- **No zero-address check for `_token`:** deploying with `_token = address(0)` would make the pool unusable.
- **Fee bounds not enforced:** unlike `setFees()`, the constructor does not restrict `_stakingFee` and `_unstakingFee` to 10%, allowing out-of-policy values at deploy time.
- **Token mutability:** `token` is only set once in the constructor and should be declared `immutable` for gas savings and safety.
- **Pause policy not defined:** contract is deployed unpaused, which may expose it before configuration.
- **SafeERC20 not declared:** the constructor assigns `IERC20` directly; without enabling SafeERC20 wrappers, token transfers rely on raw `transfer/transferFrom` behavior.

**Proof of Concept:** The following tests demonstrate how the constructor currently permits unsafe or out-of-policy deployments:

PoC A – Deployment with Zero Address This test shows that the constructor allows setting `_token = address(0)`, which causes all staking interactions to fail:

```
    function test_Construct_ZeroToken_BreaksStake() public {
```

```
        pool = new StakingPool(address(0), 5, 5);
        vm.expectRevert();
        pool.stake(1);
    }
```

PoC B – Bypassing Fee Limits at Construction This test shows that the constructor accepts staking/unstaking fees greater than the documented 10% cap, whereas `setFees()` enforces the limit:

```
    function test_Construct_AllowsFeesOverPolicy() public {
        pool = new StakingPool(address(token), 50, 50);
        vm.expectRevert(StakingPool.InvalidFees.selector);
        pool.setFees(11, 11);
    }
```

Note: These constructor gaps could allow a misconfigured or malicious deployment that is later impossible to update safely.

**Impact**
- If `_token` is set to the zero address, all ERC20 interactions revert and user funds may be locked.
- If fees are set above the intended bounds at deploy, users can be unfairly charged.
- The other issues (immutability, pause policy, SafeERC20 usage) are not direct vulnerabilities but reduce clarity, gas efficiency, and defense-in-depth.

**Recommended Mitigation:**
- Validate `_token != address(0)` in the constructor.
- Enforce the same 10% fee bounds at deployment as in `setFees()`.
- Mark `token` as `immutable`.
- Decide whether to deploy the contract paused, and document the policy.
- Use OpenZeppelin's `SafeERC20` library for all token interactions.

### [M-2] Unsafe ERC20 Operations in `stake()` and `unstake()`

**Description** The contract uses raw ERC20 calls (`transferFrom` and `transfer`) without safety wrappers or accounting for non-standard ERC20 behaviors:

`stake()` calls:

```
if (!token.transferFrom(msg.sender, address(this), _amount)) revert InvalidTransferFrom();
```

`unstake()` calls:

```
if (!token.transfer(msg.sender, unstakingAmount)) revert InvalidTransfer();
```

Two key issues arise: 1. Non-standard ERC20s: Tokens such as USDT do not return a boolean, causing raw calls that expect `true/false` to revert.

2. Fee-on-transfer or rebasing tokens: The contract assumes `_amount` or `unstakingAmount` is fully transferred, without verifying the actual balance

change. Tokens that deduct fees or rebase balances will cause internal accounting (`stakedTokens`, `totalStakes`) to diverge from the true token balance, enabling unfair reward distribution or loss of funds.

**Impact** - Incompatibility with widely-used ERC20 variants (USDT, fee-on-transfer tokens). - Internal accounting discrepancies can let users over-claim rewards or destabilize the pool.

**Proof of Concept:** This PoC demonstrates how using a fee-on-transfer token (e.g., one that burns or deducts a fee from the transferred amount) breaks the internal accounting logic of the `stake()` and `unstake()` functions in `StakingPool`.

PoC A – Fee-on-Transfer ERC20 Token Simulation This ERC20 mock simulates a 10% fee on every transfer to represent common real-world tokens like SafeMoon, etc:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.28;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract FeeOnTransferERC20 is ERC20 {
    uint256 public immutable feeBps; // e.g. 200 = 2%

    constructor(string memory name_, string memory symbol_, uint256 feeBps_)
        ERC20(name_, symbol_)
    {
        feeBps = feeBps_;
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function _update(address from, address to, uint256 value) internal override {
        if (from == address(0) || to == address(0) || value == 0) {
            super._update(from, to, value);
            return;
        }

        uint256 fee = (value * feeBps) / 10_000;
        uint256 sendAmount = value - fee;

        super._update(from, address(this), fee);
        super._update(from, to, sendAmount);
    }
}
```

PoC B – Staking With Fee-on-Transfer Token Breaks Accounting This test shows that staking 100 tokens results in fewer tokens actually received by the pool, violating internal accounting assumptions and causing reward miscalculations:

```
function test_StakeWithFeeOnTransfer_BreaksAccounting() public {
    FeeOnTransferMock feeToken = new FeeOnTransferMock();
    pool = new StakingPool(address(feeToken), 5, 5);
    feeToken.approve(address(pool), 100);
    feeToken.transfer(address(1), 100);
    vm.prank(address(1));
    feeToken.approve(address(pool), 100);

    vm.expectRevert();
    pool.stake(100);
}
```

**Recommended Mitigation:** - Use OpenZeppelin's SafeERC20 library (safeTransfer, safeTransferFrom) for all token transfers.

- Compare the contract's token balance before and after a transfer to determine the actual amount moved, and use that value for accounting.

### [M-3] Fee-Order Bug Allows Last Staker to Exit Without Paying Fee

**Description** In the `unstake()` function, the fee for unstaking is only applied if `totalStakes > 0`. However, `totalStakes` is reduced before the fee is calculated:

```
totalStakes = totalStakes - _amount;

uint256 _fee;
if (totalStakes > 0) {
    _fee = (_amount * unstakingFee) / 100;
    _addPayout(_fee);
}
```

If the user is the last staker, their `_amount` will reduce `totalStakes` to zero before fee calculation. This bypasses the fee logic entirely — they exit the pool without paying the configured fee.

This violates the intended fee policy and could be abused (e.g., stake + unstake loop to avoid paying fees).

**Impact** - Allows last staker to withdraw funds without paying the fee. - Bypasses project revenue assumptions or reward redistribution logic. - If multiple users coordinate exits, this could lead to unfair exits or reward manipulation.

**Proof of Concept** 1. Alice is the only staker with 100 tokens. 2. She calls `unstake(100)`. 3. `totalStakes` is reduced to zero before fee logic runs. 4. The

`if (totalStakes > 0)` check fails, so `_fee = 0`. 5. Alice receives the full 100 tokens instead of paying 5% fee.

**Recommended Mitigation**

- Reorder operations so the fee is calculated before `totalStakes` is reduced.
- Or, apply the fee unconditionally (e.g., `if (_amount > 0)`).

Example fix:

```
uint256 _fee = (_amount * unstakingFee) / 100;
_addPayout(_fee);

totalStakes = totalStakes - _amount;
```

# Low

No low severity issues were found in the protocol.

## Informational

### [I-1] Multiple Pragma Versions / Known Compiler Issues

**Description** The project uses different Solidity pragmas directives: - `0.8.28` (StakingPool, TestERC20) - `^0.8.20` (OpenZeppelin Ownable, ERC20, Context, Pausable) - `>=0.6.2` (IERC20Metadata) - `>=0.4.16` (IERC20) Slither flags these because some of these ranges have known historical compiler issues.

**Impact** No direct vulnerability in the current build but inconsistent pragmas reduce clarity and could, in some build setups, allow compilation with buggy compiler versions.

**Recommended Mitigation:** - Standardise pragma declarations where possible (e.g., lock all contracts to `0.8.29`). - Use a compiler version known to be safe (`>=0.8.20.1` and up).

### [I-2] Centralisation Risk: Owner Privileges

**Description** The `StakingPool` contract inherits from OpenZeppelin's `Ownable`, granting the owner privileged rights. Specifically, the owner can: - Pause and unpause the contract (`pause()`, `unpause()`) - Set staking and unstaking fees (`setFees()`) In addition, the `TestERC20` contract allows the owner to mint unlimited tokens. These centralisation points require trust that the owner will not abuse privileges or act maliciously.

**Impact** Not a direct vulnerability, but introduces trust assumptions: - The owner could set fees to the maximum allowed (10%), pause withdrawals, or mint tokens at will in `TestERC20`. - Users must fully trust the owner to act in their best interests.

**Recommended Mitigation:** - Document the role and powers of the owner for transparency. - Consider transferring ownership to a DAO, or timelock contract for reduced centralisation risk.

### [I-3] Local Variable Shadows State Variable

**Description** A local variable named `round` is declared that shadows the state variable `round` in `StakingPool` contract. Can make the code harder to read and reason about.

**Impact** No security risk, but reduces code clarity and increases the chance of developer mistakes during maintenance or upgrades.

**Recommended Mitigation:** Rename the local variable to avoid shadowing the state variable (e.g. `currentRound`).

### [I-4] No Event Emitted on Fee Updates

**Description** Both the constructor and the `setFees()` function assign values to critical parameters (`stakingFee` and `unstakingFee`) without emitting an event. Without an event, off-chain indexers and monitoring tools cannot easily detect when fees are set or changed, reducing transparency for users.

```
// Constructor
constructor(address _token, uint8 _stakingFee, uint8 _unstakingFee) Ownable(msg.sender) {
    token = IERC20(_token);
    stakingFee = _stakingFee;
    unstakingFee = _unstakingFee;
    // No event emitted here
}

// setFees()
function setFees(uint8 _stakingFee, uint8 _unstakingFee) external onlyOwner {
    if (_stakingFee > 10 || _unstakingFee > 10) revert InvalidFees();
    stakingFee = _stakingFee;
    unstakingFee = _unstakingFee;
    // No event emitted here either
}
```

**Impact** No direct on-chain vulnerability, but off-chain systems and user interfaces cannot react to fee initialisation or changes, reduces transparency and observability.

**Recommended Mitigation:** Emit an event whenever fees are set or updated, for example:

```
event FeesUpdated(uint8 stakingFee, uint8 unstakingFee);
```

and trigger it in both the constructor and `setFees()`.

### [I-5] Public Functions Should be Declared EXternal

**Description** Several functions are declared as public but are never used internally within their contracts. In Solidity, such functions should be marked external for minor gas optimizations and to better reflect intended usage. Instances include: - `pause()` in `StakingPool` - `unpause()` in `StakingPool` - `mint()` in `TestERC20`

**Impact** No security implications. However, declaring unused public functions as external: - Slightly reduces gas cost when called externally. - Clarifies the contract design intent (these functions are not for internal use).

**Recommended Mitigation:** Change the visibility from `public` to `external`.

### [I-6] Uninitialized Local Variable: `_fee`

**Description** The variable `_fee` is declared without initialization in both `stake()` and `unstake()`:

```
uint256 _fee;
```

Although Solidity defaults uninitialized `uint256` variables to zero, leaving it implicit can obscure the logic, especially for reviewers or developers unfamiliar with the default behavior. This is more of a style/readability concern than a functional issue.

**Impact** None in this context — the variable is safely assigned before use. However, in more complex functions or projects with stricter linting/static analysis, this could lead to misunderstandings or warnings.

**Recommended Mitigation** Initialize `_fee` explicitly as `uint256 _fee = 0;` to improve clarity and silence linters or static analyzers.

### [I-7] Use of Magic Number in Fee Calculation

**Description** The value 100 is hardcoded in both the stake() and unstake() functions when calculating staking and unstaking fees:

```
_fee = (_amount * stakingFee) / 100;
```

Using magic numbers — literal values without named constants — reduces code readability and maintainability. Developers reviewing the contract must infer the purpose of `100`, whereas a named constant (e.g., `FEE_DENOMINATOR`) would make the intention clearer.

**Impact** No functional impact, but impairs readability and clarity. If the denominator ever changes (e.g., switching to basis points or finer granularity), multiple updates will be needed.

**Recommended Mitigation** Define a constant at the contract level:

```
uint256 private constant FEE_DENOMINATOR = 100;
```

Then use it in fee calculations:

```
_fee = (_amount * stakingFee) / FEE_DENOMINATOR;
```

### [I-8] `unstake()` Allowed While Contract is Paused

**Description** The `unstake()` function does not use the `whenNotPaused` modifier, meaning users can withdraw their tokens even when the contract is paused:

```
function unstake(uint256 _amount) external {
    ...
}
```

In contrast, `stake()` and `donateToPool()` are protected with `whenNotPaused`. This may be intentional — e.g., to allow withdrawals during emergency pauses — but this divergence should be explicitly documented.

**Impact** If unintended, this allows users to unstake even when other functionality is paused, potentially bypassing protocol-level restrictions or protections.

**Recommended Mitigation**

If this is intended, document the design rationale in the NatSpec or external documentation.

If not intended, add `whenNotPaused` to `unstake()` for consistency with other functions.

### [I-9] Use of uint Instead of uint256

**Description** The function `getPendingReward()` uses the generic `uint` type instead of the explicitly sized `uint256`:

```
uint stakerRound = stakers[_staker].round;
```

Although `uint` is an alias for `uint256` in Solidity, it's considered best practice to explicitly use `uint256` throughout for clarity, gas modeling consistency, and compatibility with static analysis tools.

**Impact** No functional impact, but inconsistency in type usage can reduce code readability and raise static analysis/linting flags.

**Recommended Mitigation** Change the declaration to:

```
uint256 stakerRound = stakers[_staker].round;
```