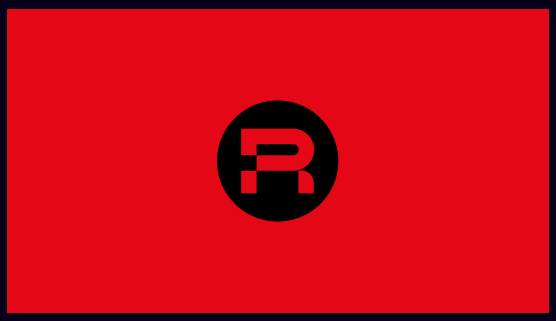


SMART CONTRACTS REVIEW

zokyo

January 27th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING $REAL

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 1 Low issue: 1 resolved = 0 points deducted
- 1 Informational issue: 1 resolved = 0 points deducted

Thus, the score is 100

TECHNICAL SUMMARY

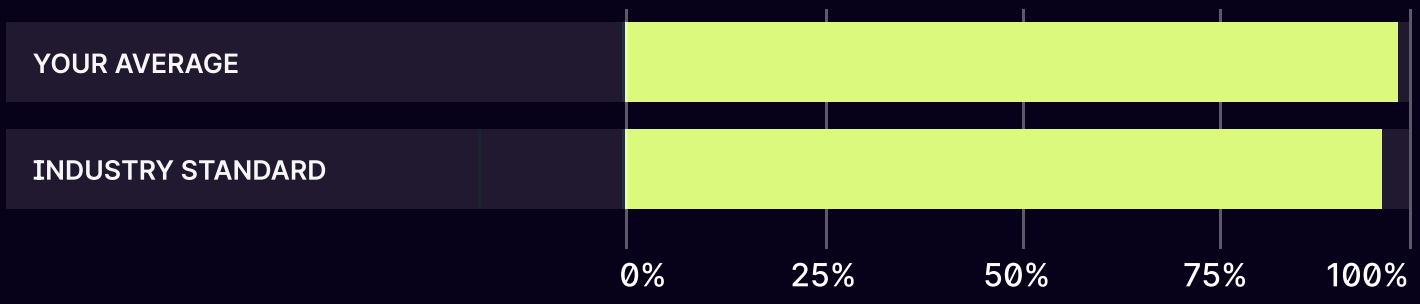
This document outlines the overall security of the $REAL smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the $REAL smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



94.67% of the codebase is covered by tests.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the $REAL team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the $REAL repository:
Repo: https://github.com/bltzr-gg/realbet/tree/master/apps/evm-contracts

Last commit - da2cca37c1c1b5beff4db428b39456a17a2d5d56

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- TokenStaking.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of $REAL smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The TokenStaking contract is a staking system that extends the ERC20 token standard while incorporating features such as epoch-based rewards, tiered staking, and governance integration. Built using OpenZeppelin's libraries, it employs ReentrancyGuard and AccessControl for security and role-based permissions. The contract allows users to stake a specified ERC20 token (TOKEN) into predefined tiers, each with a unique lock period and multiplier. These tiers determine the "effective amount" of tokens, which influences reward distribution. Rewards are distributed on an epoch basis and can be configured dynamically by authorized roles. The contract includes functionality to stake tokens, claim rewards, and unstake after the lock period. It tracks the total effective supply over time and adjusts rewards accordingly. A voting mechanism is integrated to encourage user participation in governance.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the $REAL team and the $REAL team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Logical Error in Transfer Validation	High	Resolved
2	Reward Claiming Fails Due to Invalid Epoch Handling	High	Resolved
3	Rewards Loss on Unstaking	Medium	Resolved
4	Users Cant Stake For Epoch 0	Medium	Resolved
5	Updated Tier State Should Not Be Applied For Pre - Update Stakes	Medium	Resolved
6	Missing Event Emission in Administrative Functions	Low	Resolved
7	Lock Period Validation Logic Duplication	Informational	Resolved

HIGH-1 | RESOLVED

Logical Error in Transfer Validation

The `_update` function checks if both `from` and `to` are `address(0)` to disallow transfers, which is logically incorrect. Therefore, A user could still burn their tokens.

```
function _update(address from, address to, uint256 value) internal override {
// End-users cannot transfer or burn their tokens
if (from != address(0) && to != address(0))
{
revert TransferNotAllowed();
 }
```

Recommendation:
Update the condition in `_update`.

HIGH-2 | RESOLVED

Reward Claiming Fails Due to Invalid Epoch Handling

The TokenStaking contract's reward claiming functionality is failing due to improper handling of epoch validation. Specifically, the contract is rejecting attempts to claim rewards for valid epochs, throwing an InvalidEpoch() error.

Steps to Reproduce:
1. Deploy the TokenStaking contract
2. Stake tokens for a user
3. Set rewards for future epochs
4. Move time forward to a future epoch
5. Attempt to claim rewards for epochs after the stake was created

Expected Behavior:
The contract should allow claiming rewards for all epochs after the stake was created, starting from the epoch immediately following the stake creation.

Actual Behavior:
The contract throws an InvalidEpoch() error when attempting to claim rewards, even for seemingly valid epochs.

Potential Cause:
The issue likely stems from how the contract determines valid epochs for reward claiming. It appears that the contract is incorrectly validating the epochs passed to the calculateRewards function.

Relevant Code:

```
function calculateRewards(
    uint256 stakeIndex,
    uint32[] calldata epochs,
    bytes32[][] calldata merkleProofs
) public view returns (uint256) {
    if (stakeIndex >= userStakes[msg.sender].length) {
        revert InvalidStakeIndex();
    }
    Stake memory userStake = userStakes[msg.sender][stakeIndex];

    uint256 reward = 0;
    uint256 lastEpoch = getCurrentEpoch();

    uint256 lastProcessedEpoch = userStake.lastClaimEpoch;

    for (uint256 i = 0; i < epochs.length; i++) {
        uint256 epoch = epochs[i];
```

```
        // Ensure epochs are in ascending order and not repeated
        if (epoch <= lastProcessedEpoch || epoch >= lastEpoch) {
            revert InvalidEpoch();
        }
        // ... (rest of the function)
    }

    return reward;
}
```

Potential Fix:
Review and adjust the epoch validation logic in the calculateRewards function. Ensure that it correctly handles the initial case where lastClaimEpoch is `0` (for newly created stakes) and allows claiming rewards for all valid epochs after stake creation.

Additional Notes:
This bug significantly impacts the core functionality of the staking contract, preventing users from claiming their rightfully earned rewards. It's crucial to address this issue promptly to ensure the proper operation of the staking system.

Proof of Concept:

```
function testClaimRewards() public {
    // Setup: Stake tokens for user1
    vm.startPrank(user1);
    realToken.mint(user1, 1e18);
    realToken.approve(address(staking), 1e18);
    staking.stake(1e18, 0);
```

```
    vm.stopPrank();


    // Verify that the stake was created
    TokenStaking.Stake[] memory stakes = staking.getUserStakes(user1);
    require(stakes.length > 0, "Stake was not created");
    console.log("Stake created for user1. Number of stakes:",
stakes.length);


    // Log stake details
    console.log("Stake amount:", stakes[0].amount);
    console.log("Stake effectiveAmount:", stakes[0].effectiveAmount);
    console.log("Stake tierIndex:", stakes[0].tierIndex);
    console.log("Stake startTime:", stakes[0].startTime);
    console.log("Stake lastClaimEpoch:", stakes[0].lastClaimEpoch);


    // Get the current epoch
    uint256 currentEpoch = staking.getCurrentEpoch();
    console.log("Current epoch:", currentEpoch);


    // Set rewards for the next four epochs
    staking.setRewardForEpoch(currentEpoch + 1, 100e18);
    staking.setRewardForEpoch(currentEpoch + 2, 200e18);
    staking.setRewardForEpoch(currentEpoch + 3, 300e18);
    staking.setRewardForEpoch(currentEpoch + 4, 400e18);
```

```
// Set merkle roots for the next four epochs
bytes32 merkleRoot = keccak256(abi.encodePacked(user1));
staking.setMerkleRoot(uint32(currentEpoch + 1), merkleRoot);
staking.setMerkleRoot(uint32(currentEpoch + 2), merkleRoot);
staking.setMerkleRoot(uint32(currentEpoch + 3), merkleRoot);
staking.setMerkleRoot(uint32(currentEpoch + 4), merkleRoot);

// Move to epoch 5
vm.warp(block.timestamp + 4 * staking.epochDuration());
currentEpoch = staking.getCurrentEpoch();
console.log("New current epoch:", currentEpoch);

// Set up epochs and merkle proofs
uint32[] memory epochs = new uint32[](4);
epochs[0] = uint32(currentEpoch - 3);
epochs[1] = uint32(currentEpoch - 2);
epochs[2] = uint32(currentEpoch - 1);
epochs[3] = uint32(currentEpoch);
bytes32[][] memory merkleProofs = new bytes32[][](4);
merkleProofs[0] = new bytes32[](0);
merkleProofs[1] = new bytes32[](0);
merkleProofs[2] = new bytes32[](0);
```

```
    merkleProofs[3] = new bytes32[](0);


    console.log("Calculating rewards for epochs:");

    console.log(epochs[0], epochs[1]);

    console.log(epochs[2], epochs[3]);


    // Calculate expected reward

    vm.prank(user1);

    uint256 expectedReward = staking.calculateRewards(0, epochs,
merkleProofs);

    console.log("Expected reward:", expectedReward);


    require(expectedReward > 0, "Expected reward should be greater than
0");


    // Record user's balance before claiming

    uint256 balanceBefore = realToken.balanceOf(user1);


    // Claim rewards

    vm.prank(user1);

    vm.expectEmit(true, true, true, true);

    emit TokenStaking.RewardClaimed(user1, expectedReward);

    staking.claimRewards(0, epochs, merkleProofs);
```

```
    // Verify reward transfer
    uint256 balanceAfter = realToken.balanceOf(user1);
    assertEq(balanceAfter - balanceBefore, expectedReward, "Incorrect
reward amount transferred");

    // Verify lastClaimEpoch update
    (,,,,uint32 lastClaimEpoch) = staking.userStakes(user1, 0);
    assertEq(lastClaimEpoch, epochs[3], "Last claim epoch not updated
correctly");

    // Attempt to claim rewards again (should result in 0 reward)
    vm.prank(user1);
    uint256 secondClaimReward = staking.calculateRewards(0, epochs,
merkleProofs);
    assertEq(secondClaimReward, 0, "Second claim should result in 0
reward");
}
```

```
[FAIL: InvalidEpoch()] testClaimRewards() (gas: 493870)
Logs:
  Stake created for user1. Number of stakes: 1
  Stake amount: 1000000000000000000
  Stake effectiveAmount: 100000000000000000
  Stake tierIndex: 0
  Stake startTime: 604801
```

```
  Stake lastClaimEpoch: 0

  Current epoch: 1

  New current epoch: 5

  Calculating rewards for epochs:

  2 3

  4 5


Traces:

  [493870] TokenStakingTest::testClaimRewards()

    ├─ [0] VM::startPrank(ECRecover:
[0x0000000000000000000000000000000000000001])

    |   └─ ← [Return]

    ├─ [12694] 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f::mint(ECRecover:
[0x0000000000000000000000000000000000000001], 1000000000000000000 [1e18])

    |   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000,
to: ECRecover: [0x0000000000000000000000000000000000000001], value:
1000000000000000000 [1e18])

    |   └─ ← [Stop]

    ├─ [7634]
0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f::approve(0x2e234DAe75C793f67A35
089C9d99245E1C58470b, 1000000000000000000 [1e18])

    |   ├─ emit Approval(owner: ECRecover:
[0x0000000000000000000000000000000000000001], spender:
0x2e234DAe75C793f67A35089C9d99245E1C58470b, value: 1000000000000000000
[1e18])

    |   └─ ← [Return] true
```

```
    ├─ [228439]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::stake(1000000000000000000
[1e18], 0)

    │   ├─ [8957]
0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f::transferFrom(ECRecover:
[0x0000000000000000000000000000000000000001],
0x2e234DAe75C793f67A35089C9d99245E1C58470b, 1000000000000000000 [1e18])

    │   │   ├─ emit Transfer(from: ECRecover:
[0x0000000000000000000000000000000000000001], to:
0x2e234DAe75C793f67A35089C9d99245E1C58470b, value: 1000000000000000000
[1e18])

    │   │   └─ ← [Return] true

    │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000,
to: ECRecover: [0x0000000000000000000000000000000000000001], value:
1000000000000000000 [1e18])

    │   ├─ emit Staked(user: ECRecover:
[0x0000000000000000000000000000000000000001], amount: 1000000000000000000
[1e18], tierIndex: 0)

    │   └─ ← [Stop]

    ├─ [0] VM::stopPrank()

    │   └─ ← [Return]

    ├─ [1800]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::getUserStakes(ECRecover:
[0x0000000000000000000000000000000000000001]) [staticcall]

    │   └─ ← [Return] [Stake({ amount: 1000000000000000000 [1e18],
effectiveAmount: 100000000000000000 [1e17], tierIndex: 0, startTime:
604801 [6.048e5], lastClaimEpoch: 0 })]

    ├─ [0] console::log("Stake created for user1. Number of stakes:", 1)
[staticcall]

    │   └─ ← [Stop]
```

```
    ├─ [0] console::log("Stake amount:", 1000000000000000000 [1e18])
[staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("Stake effectiveAmount:", 100000000000000000
[1e17]) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("Stake tierIndex:", 0) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("Stake startTime:", 604801 [6.048e5]) [staticcall]
    │   └─ ← [Stop]
    ├─ [0] console::log("Stake lastClaimEpoch:", 0) [staticcall]
    │   └─ ← [Stop]
    ├─ [596] 0x2e234DAe75C793f67A35089C9d99245E1C58470b::getCurrentEpoch()
[staticcall]
    │   └─ ← [Return] 1
    ├─ [0] console::log("Current epoch:", 1) [staticcall]
    │   └─ ← [Stop]
    ├─ [26574]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setRewardForEpoch(2,
10000000000000000000 [1e20])
    │   ├─ emit RewardSet(epoch: 2, amount: 10000000000000000000 [1e20])
    │   └─ ← [Stop]
    ├─ [24574]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setRewardForEpoch(3,
20000000000000000000 [2e20])
    │   ├─ emit RewardSet(epoch: 3, amount: 20000000000000000000 [2e20])
```

```
    |   └─ ← [Stop]

    ├─ [24574]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setRewardForEpoch(4,
30000000000000000000 [3e20])

    |   ├─ emit RewardSet(epoch: 4, amount: 30000000000000000000 [3e20])

    |   └─ ← [Stop]

    ├─ [24574]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setRewardForEpoch(5,
40000000000000000000 [4e20])

    |   ├─ emit RewardSet(epoch: 5, amount: 40000000000000000000 [4e20])

    |   └─ ← [Stop]

    ├─ [24132]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setMerkleRoot(2,
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   ├─ emit MerkleRootSet(epoch: 2, merkleRoot:
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   └─ ← [Stop]

    ├─ [24132]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setMerkleRoot(3,
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   ├─ emit MerkleRootSet(epoch: 3, merkleRoot:
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   └─ ← [Stop]

    ├─ [24132]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setMerkleRoot(4,
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   ├─ emit MerkleRootSet(epoch: 4, merkleRoot:
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    |   └─ ← [Stop]
```

```
    ├─ [24132]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::setMerkleRoot(5,
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    │   ├─ emit MerkleRootSet(epoch: 5, merkleRoot:
0x1468288056310c82aa4c01a7e12a10f8111a0560e72b700555479031b86c357d)

    │   └─ ← [Stop]

    ├─ [361] 0x2e234DAe75C793f67A35089C9d99245E1C58470b::epochDuration()
[staticcall]

    │   └─ ← [Return] 604800 [6.048e5]

    ├─ [0] VM::warp(3024001 [3.024e6])

    │   └─ ← [Return]

    ├─ [596] 0x2e234DAe75C793f67A35089C9d99245E1C58470b::getCurrentEpoch()
[staticcall]

    │   └─ ← [Return] 5

    ├─ [0] console::log("New current epoch:", 5) [staticcall]

    │   └─ ← [Stop]

    ├─ [0] console::log("Calculating rewards for epochs:") [staticcall]

    │   └─ ← [Stop]

    ├─ [0] console::log(2, 3) [staticcall]

    │   └─ ← [Stop]

    ├─ [0] console::log(4, 5) [staticcall]

    │   └─ ← [Stop]

    ├─ [0] VM::prank(ECRecover:
[0x0000000000000000000000000000000000000001])

    │   └─ ← [Return]
```

```
    ├─ [6501]
0x2e234DAe75C793f67A35089C9d99245E1C58470b::calculateRewards(0, [2, 3, 4,
5], [[], [], [], []]) [staticcall]
    │   └─ ← [Revert] InvalidEpoch()
    └─ ← [Revert] InvalidEpoch()
```

Recommendation:

1. Thoroughly review the epoch validation logic in the calculateRewards function.
2. Consider adding more detailed error messages to help diagnose specific validation failures.
3. Implement comprehensive unit tests covering various epoch scenarios, including edge cases.
4. After fixing, conduct a thorough review of all functions that interact with epochs to ensure consistent behavior across the contract.

MEDIUM-1 | RESOLVED

Rewards Loss on Unstaking

If a user unstakes their tokens before claiming rewards, the `_claimRewards` function is not invoked, and the user loses unclaimed rewards. This could lead to significant user dissatisfaction and financial loss.

Recommendation:

Include a check in the `unstake` function to ensure all rewards are claimed before allowing unstaking. Alternatively, automatically claim rewards during the unstaking process.

Users Cant Stake For Epoch 0

When staking the following code is executed to push the stake into the userStakes array →

```
userStakes[msg.sender].push(
            Stake({
                amount: amount,
                effectiveAmount: effectiveAmount,
                tierIndex: tierIndex,
                startTime: uint32(block.timestamp),
                lastClaimEpoch: uint32(currentEpoch - 1)
            })
        );
```

But if the current epoch is `0` , the currentEpoch - 1 line of code would revert resulting in users not being able to stake in the first epoch . Also , from the code it is evident that users are intended to stake at the `0`th epoch i.e. in updateTotalEffectiveSupply function it is checked if current epoch > `0` , indicating that epoch `0` is intended to hold an effective supply.

Recommendation:
For epoch `0` assign `0` as the lastClaimEpoch.

MEDIUM-3 | RESOLVED

Updated Tier State Should Not Be Applied For Pre - Update Stakes

UserA stakes at time period t and chose the tierIndex = 1 to stake , this was because tier1 seemed most appropriate to user1 , after some time the admin decides to update the tier1 and thus calls setTier() with a new set of lockPeriod and multiplier . This new set of values might be too high/low for the original user and when checking values , for example when unstaking →

```
if (block.timestamp < userStake.startTime +
tiers[userStake.tierIndex].lockPeriod) {
            revert LockPeriodNotEnded();
        }
```

The new updated lockPeriod would be used instead of the one that was assigned when user staked , this can be non-intentional for the user.

Recommendation:
Use the original stake information instead for users staked pre - update.

LOW-1 | RESOLVED

Missing Event Emission in Administrative Functions

The `setEpochDuration` and `setDefaultEpochRewards` functions allow administrators to modify critical parameters of the staking mechanism, but they do not emit any events when invoked.

Recommendation:
Add event emissions to these functions to log parameter updates.

Lock Period Validation Logic Duplication

The `unstake` function manually checks the lock period using an inline condition instead of reusing the `isLocked` function. This leads to redundant code and increased maintenance overhead.

Recommendation:
Replace the inline condition with a call to `isLocked`.

	TokenStaking.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting $REAL in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the $REAL contract requirements for details about issuance amounts and how the system handles these.

Ran 28 tests for test/TokenStaking.t.sol:TokenStakingTest
[PASS] testCalculateRewardsInvalidEpoch() (gas: 297700)
[PASS] testCalculateRewardsInvalidStakeIndex() (gas: 315777)
[PASS] testCalculateRewardsWithVoting() (gas: 318887)
[PASS] testCannotSetRewardForPastEpoch() (gas: 22604)
[PASS] testCannotStakeZeroAmount() (gas: 17537)
[PASS] testClaimRewardsInvalidStakeIndex() (gas: 322173)
[PASS] testDeployment() (gas: 17626)
[PASS] testGetCurrentEpoch() (gas: 17781)
[PASS] testGetRewardsForEpoch() (gas: 43843)
[PASS] testGetRewardsForEpochFallback() (gas: 86127)
[PASS] testGetTotalEffectiveSupplyAtEpochForFilledEpochs() (gas: 417346)
[PASS] testGetTotalEffectiveSupplyAtEpochForFutureEpochs() (gas: 281514)
[PASS] testIsLockedInvalidStakeIndex() (gas: 275684)
[PASS] testSetDefaultEpochRewards() (gas: 48182)
[PASS] testSetEpochDuration() (gas: 17586)
[PASS] testSetRewardEmitsEvent() (gas: 43319)
[PASS] testSetRewardForCurrentEpoch() (gas: 43230)
[PASS] testSetRewardForFutureEpoch() (gas: 43449)
[PASS] testSetTier() (gas: 66212)
[PASS] testSetTierAddNew() (gas: 115010)
[PASS] testSetTierEmitTierAdded() (gas: 98162)
[PASS] testSetTierEmitTierUpdated() (gas: 29110)
[PASS] testSetTierUpdateExisting() (gas: 29513)
[PASS] testSetTierZeroMultiplier() (gas: 12493)
[PASS] testStakeInvalidTierIndex() (gas: 95299)
[PASS] testTransferNotAllowed() (gas: 279181)
[PASS] testUnstakeInvalidStakeIndex() (gas: 286718)
[PASS] testUnstakeLockPeriodNotEnded() (gas: 465736)
Suite result: ok. 28 passed; 0 failed; 0 skipped; finished in 50.91ms (34.92ms CPU time)

Ran 18 tests for test/TokenStakingFuzz.t.sol:TokenStakingFuzzTest
[PASS] testFuzzCannotSetRewardForPastEpoch(uint256,uint256) (runs: 259, μ: 24465, ~: 24465)
[PASS] testFuzzCannotStakeZeroAmount(uint32) (runs: 257, μ: 53028, ~: 53028)
[PASS] testFuzzDeployment(address,address) (runs: 265, μ: 3282945, ~: 3282945)
[PASS] testFuzzGetCurrentEpoch(uint256) (runs: 261, μ: 19163, ~: 19163)
[PASS] testFuzzGetRewardsForEpoch(uint256) (runs: 264, μ: 44672, ~: 44672)
[PASS] testFuzzGetRewardsForEpochFallback(uint256,uint256) (runs: 264, μ: 85264, ~: 85264)
[PASS] testFuzzSetDefaultEpochRewards(uint256) (runs: 265, μ: 40408, ~: 40408)
[PASS] testFuzzSetEpochDuration(uint256) (runs: 261, μ: 18225, ~: 18225)
[PASS] testFuzzSetRewardEmitsEvent(uint256) (runs: 264, μ: 44093, ~: 44093)
[PASS] testFuzzSetRewardForCurrentEpoch(uint256) (runs: 264, μ: 44516, ~: 44516)
[PASS] testFuzzSetRewardForFutureEpoch(uint256,uint256) (runs: 259, μ: 45346, ~: 45346)
[PASS] testFuzzSetTier(uint256,uint256,uint256) (runs: 263, μ: 125090, ~: 126232)
[PASS] testFuzzSetTierAddNew(uint256,uint256) (runs: 261, μ: 117399, ~: 117399)
[PASS] testFuzzSetTierEmitTierAdded(uint256,uint256) (runs: 261, μ: 99096, ~: 99096)
[PASS] testFuzzSetTierEmitTierUpdated(uint256,uint256,uint256) (runs: 265, μ: 71249, ~: 70990)
[PASS] testFuzzSetTierUpdateExisting(uint256,uint256) (runs: 261, μ: 60459, ~: 60470)
[PASS] testFuzzSetTierZeroMultiplier(uint256,uint256) (runs: 263, μ: 18200, ~: 18329)
[PASS] testFuzzUnstakeInvalidStakeIndex(uint256) (runs: 265, μ: 321355, ~: 321355)

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
TokenStaking.sol	94.67	91.30	100	95.45
All Files	**94.67**	**91.30**	**100**	**95.45**

We are grateful for the opportunity to work with the $REAL team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the $REAL team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo