QuillAudits

AUDIT REPORT

November 2025

For

Yieldra

Table of Content

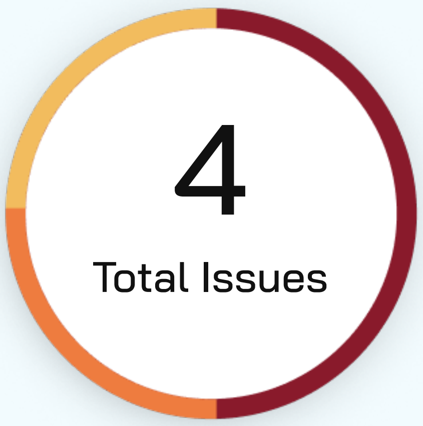
Executive Summary

Project Name	Yieldra
Project URL	https://yieldra.io/
Overview	The YieldraStaking contract is an epoch-based staking system that allows users to lock a single stake per address to earn rewards from a funded reward pool, with each epoch lasting seven days. Rewards are distributed based on a fixed percentage (rewardPerEpochBPS) of the remaining pool, and staking power grows exponentially with the chosen duration, calculated as amount * (1 + r)^epochLength. A small deposit fee and early-unstake penalties are split between the protocol and treasury, incentivizing longer commitments. Users can claim rewards only for completed epochs, and all transfers use SafeERC20 with ReentrancyGuard protection. The owner manages reward funding, treasury updates, and can retire the system by refunding unallocated rewards.
Source Code link	https://github.com/Yieldra-Protocol/staking-contracts/tree/main/src
Branch	main
Contracts in Scope	src/YieldraStaking.sol src/WadRayMath.sol
Commit Hash	0500b8109e48ad953a8451e0a0dec8f69d0df441
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	27th October 2025 - 5th November 2025
Updated Code Received	6th November 2025
Review 2	6th November 2025 - 11th November 2025
Fixed In	d1f76b256a2119690a9f2188bd0ed503258c09b7

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	2 (50%)
🟥	High	0 (0%)
🟧	Medium	1 (25%)
🟨	Low	1 (25%)
🟪	Informational	0 (0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged (Intended Design, not a bug as per Yieldra team)	1	0	1	1	0
Partially Resolved	0	0	0	0	0
Resolved	1	0	0	0	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Reward Over-Distribution Due to Power Calculation with Updated Stake	Critical	Resolved
2	Epoch Snapshot Mutation During Unstake Causes Reward Over-Distribution	Critical	Acknowledged (Intended Design, not a bug as per Yieldra team)
3	Reward Distribution Manipulation via setRewardPerEpochBPS	Medium	Acknowledged (Intended Design, not a bug as per Yieldra team)
4	Redundant Storage Reads in Loops	Low	Acknowledged (Intended Design, not a bug as per Yieldra team)

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw
- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Likelihood \ Impact	🟥 High	🟧 Medium	🟨 Low
🟥 High	Critical	High	Medium
🟧 Medium	High	Medium	Low
🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Reward Over-Distribution Due to Power Calculation with Updated Stake

Resolved

Path

contracts/YieldraStaking.sol

Function Name

`increaseStake()`

Description

The increaseStake() function allows users to add more tokens to their existing stake and increases their staking power accordingly. However, a critical flaw in the reward calculation logic allows users to claim inflated rewards for past epochs using their updated (increased) power instead of their historical power at the time those epochs occurred.

Root Cause

When a user calls increaseStake() during an ongoing epoch:

1. _rollEpochs() is called, which snapshots the current epoch's eligiblePowerRay with all active stakers' original power
2. _settleMatured() force-claims all completed epochs (up to currentEpoch - 1) using the user's old power
3. The user's stake power (s.powerRay) is then updated to the new increased value
4. However, the current ongoing epoch (e.g., epoch 2) is NOT claimed during increaseStake()
5. Later, when the user calls claim() after that epoch completes, the reward calculation uses the updated s.powerRay from storage instead of the power that existed when the epoch was active

Impact

Users can claim more tokens than their allocation.

Likelihood: High

Recommendation

Modify increaseStake() to settle the current ongoing epoch before updating the user's power or Maintain a historical record of each user's power across epochs

Epoch Snapshot Mutation During Unstake Causes Reward Over-Distribution

Acknowledged
(Intended Design, not a bug as per Yieldra team)

Path
contracts/YieldraStaking.sol

Function Name
`unstake()`

Description
The unstake() and unstakeWithPenalty() functions incorrectly modify the eligiblePowerRay of an already-initialized epoch when users exit their positions. This violates a core protocol invariant that epoch snapshots must be immutable once set. By reducing the eligible power denominator after rewards have been allocated, the contract inadvertently inflates the reward share for remaining stakers, leading to systematic over-distribution of rewards.

Root Cause
When an epoch begins (via _rollEpochs()), the contract captures a snapshot of total eligible staking power and allocates rewards based on this snapshot:

```
function _rollEpochs() internal {
    for (uint256 e = lastEpochProcessed + 1; e <= currentEpoch; e++) {
        // Snapshot taken HERE
        es.eligiblePowerRay = totalActivePowerRay;

        // Rewards allocated based on THIS snapshot
        uint256 alloc = (rewardPool * rewardPerEpochBPS) / BPS_DENOMINATOR;
        es.rewardAllocated += alloc;
    }
}
```

However, when a user unstakes during this same epoch, the contract retroactively modifies the snapshot:

```
function unstake() external nonReentrant {
    // … existing logic …

    if (startEpoch <= currentEpoch) {
        EpochState storage es = epochs[currentEpoch];
        if (es.initialized && es.eligiblePowerRay >= powerRay) {
            // BUG: Modifying snapshot AFTER allocation
            es.eligiblePowerRay -= powerRay;  // ← BREAKS IMMUTABILITY
        }
    }
}
```

This creates a mismatch where:

- Numerator (allocated rewards): Based on original snapshot (all stakers)
- Denominator (eligible power): Based on modified snapshot (after unstakes)

Impact

Over allocation of tokens than intended.

Likelihood: High

Recommendation

Preserve Epoch Snapshot Immutability and remove all code that modifies eligiblePowerRay after an epoch has been initialized.

Yieldra Team's Comment

This is actually the design for reward and punishment in the Staking behavior. The current (on-going epoch) is not final until next epoch start. To be eligible a staking power must be inside the epoch from beginning until the end. Therefore, exiting an epoch (with or without penalty) will forfeit user's claim and benefit the others who still maintain their stake.

Medium Severity Issues

Reward Distribution Manipulation via setRewardPerEpochBPS

Acknowledged
(Intended Design, not a bug as per Yieldra team)

Path

contracts/YieldraStaking.sol

Function Name

`setRewardPerEpochBPS()`

Description

The setRewardPerEpochBPS function allows the contract owner to modify the reward rate (rewardPerEpochBPS) at any time without first processing pending epochs. This creates a vulnerability where reward rate changes apply retroactively to all unprocessed epochs, rather than only affecting future epochs.

When epochs accumulate without being rolled (via rollEpochs()), changing the reward rate before processing these epochs causes all pending epochs to be calculated using the new rate instead of the rate that was active when users staked. This breaks the fundamental expectation that staking rewards are determined by the rate at the time of staking or epoch occurrence.

Impact

Users may receive significantly reduced rewards.

Likelihood: High

Recommendation

Enforce automatic processing of all pending epochs before allowing any reward rate changes.

Yieldra Team’s Comment

This one is a valid concern if we want to limit how the “Program Runner/Owner” may change the rewardPerEpochBPS, but this is clearly an administrative decision from the Program Runner, they can even stop the program altogether.

Low Severity Issues

Redundant Storage Reads in Loops

Acknowledged
(Intended Design, not a bug as per Yieldra team)

Path
contracts/YieldraStaking.sol

Function Name
`rollEpochs()`

Description
The _rollEpochs and rollEpochs functions perform multiple redundant storage reads inside a loop, specifically for variables such as retired and rewardPerEpochBPS. These values remain constant during execution and do not need to be fetched from storage repeatedly.

Each iteration currently triggers multiple SLOAD operations, resulting in unnecessary gas costs, especially when processing several epochs. Cache storage variables before entering the loop to minimize redundant reads

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Constructor should initialize all default parameters correctly
- ✓ Only owner should be able to fund rewards and update treasury address
- ✓ Funding rewards should emit RewardFunded event and update reward pool balance
- ✓ Creating a stake should apply correct deposit fee and schedule staking power for next epoch
- ✓ Deposit fee should split correctly between protocol and treasury
- ✓ Stake details (amount, epochs, power) should store accurately after creation
- ✓ Increasing stake mid-epoch should not boost rewards for the current epoch
- ✓ Increasing stake should correctly update user's total amount and staking power
- ✓ Matured rewards should be automatically claimed when increasing stake
- ✓ Unstaking should fairly distribute rewards to all users based on snapshot power
- ✓ Unstaking should not increase remaining stakers' reward share
- ✓ Unstaking after maturity should return full staked amount with no penalty
- ✓ Unstaking with penalty should deduct correct fee and distribute penalty properly
- ✓ Power subtraction on unstake should not distort reward allocations in ongoing epoch
- ✓ Reward distribution across epochs should always sum up to total allocated rewards
- ✓ Claiming rewards should transfer correct reward amount to user based on power share
- ✓ Burned or ended stakes should be deleted and marked inactive
- ✓ Total active power should update correctly after stakes and unstakes
- ✓ Claiming rewards multiple times should not overpay users
- ✓ Epoch data (allocations, power snapshots) should remain consistent across operations

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
YieldraStaking	constructor	Misconfiguration of protocol/treasury/ reward token could break accounting or redirect funds; Centralization risk via single owner controlling lifecycle
	createStake	Reentrancy during token transfer; Deposit fee miscalculation causing fund loss; Malicious ERC20 token could revert or drain gas; Incorrect scheduling may allow early reward accrual
	increaseStake	Front-running to boost power after reward allocation; Reward double counting if _settleMatured misfires; Fee splitting inconsistencies; Integer rounding on weighted extension may lead to unfair boost
	unstake	Logic bypass if retired — early exits with no penalty; Improper delete could break accounting; Reentrancy via malicious ERC20 callback
	unstakeWithPenalty	Miscomputed penaltyBPS overflow; Rounding in split between protocol/ treasury can misroute funds; User grief via extreme epoch length configuration
	claim / claimUpTo	Double claim or skipped epoch risk if lastClaimedEpoch not updated atomically; Reentrancy during token payout; Epoch range manipulation to drain pool

Closing Summary

During the course of the security Audit assessment, a total of four issues were identified: two Critical, one Medium, and one Low severity finding.

The Yieldra team has already resolved one of the Critical issues related to Reward Over-Distribution. The second Critical issue, Epoch Snapshot Mutation During Unstake Leading to Reward Over-Distribution, was Discussed with the Yieldra team. They confirmed that this behavior is an intentional component of their reward and punishment model in the staking mechanism, and therefore not considered a vulnerability but part of the designed business logic.

Similarly, the Medium-severity finding Reward Distribution Manipulation via SetRewardPerEpochBPS was discussed with the Yieldra team. The program runner has full administrative authority to modify or even stop the program, and this behavior is also intended by design. As such, the issue has been acknowledged and accepted as expected business logic rather than a security flaw.

The remaining Low-severity issue related to Redundant Storage Reads in Loops has been reported for the team's consideration and does not pose immediate risk to protocol integrity.

Overall, the Yieldra Team demonstrates a reasonable security posture, with critical concerns either remediated or validated as intended logic. The team has been responsive throughout the audit process, and we recommend continuing with Multiple Security Audits and to apply best practices around access control, reward configuration, and operational security as the protocol evolves.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

Whilewe have conducted a thorough review, security is an ongoing process. Westrongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

November 2025

For

Yieldra

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com