



# AUDIT REPORT

---




September 2025

For



**WachAI**

# Table of Content

Executive Summary	03
Number of Security Issues per Severity	05
Summary of Issues	06
Checked Vulnerabilities	07
Techniques and Methods	09
Types of Severity	11
Types of Issues	12
Severity Matrix	13
 <b>Medium Severity Issues</b>	14
1. Restake During Cooldown Leads to Stuck Withdrawal	14
2. Merkle Leaf Construction Allows Cross-Round Replay Attacks	16
 <b>Low Severity Issues</b>	17
3. Users Do Not Earn Rewards During Cooldown Period	17
4. Missing Validation for Total Percentage Allocation	18
 <b>Informational Issues</b>	19
5. Floating Solidity Pragma Used	19
6. Ownable Should Use Two-Step Ownership Transfer	19
7. Confusing Naming of Withdrawal Functions	20
8. Unused EmergencyWithdrawal Event Declaration	20
Automated Tests	21
Functional Tests	21
Threat Model	22
Closing Summary & Disclaimer	23



# Executive Summary

<b>Project Name</b>	WachAi
<b>Protocol Type</b>	Staking & Airdrop
<b>Project URL</b>	<a href="https://wach.ai/">https://wach.ai/</a>
<b>Overview</b>	<p><b>Staking:</b></p> <p>The staking contract allows users to lock tokens, earn rewards over defined periods, and withdraw with a cooldown. Issues mainly relate to reward accounting during cooldown, locked rewards if added before any stake, and missing robustness in recovery logic.</p> <p><b>Airdrop:</b></p> <p>The airdrop contract distributes tokens via a Merkle tree with per-user percentage allocations. Issues include missing roundId in leaf construction, lack of total percentage validation, and an unused emergency event that may cause confusion.</p>
<b>Audit Scope</b>	The scope of this Audit was to analyze the Wach AI Smart Contracts for quality, security, and correctness.
<b>Source Code link</b>	<a href="https://github.com/quillai-network/website-staking">https://github.com/quillai-network/website-staking</a>
<b>Branch</b>	Main
<b>Contracts in Scope</b>	Staking.sol WACHAirdropDistributor.sol
<b>Commit Hash</b>	2aabe6ffbca16c8cbcb8a92c34207674b5c8fb10
<b>Language</b>	Solidity
<b>Blockchain</b>	Ethereum
<b>Method</b>	Manual Analysis, Functional Testing, Automated Testing
<b>Review 1</b>	2nd September 2025 - 9th September 2025
<b>Updated Code Received</b>	9th September 2025
<b>Review 2</b>	15th September 2025

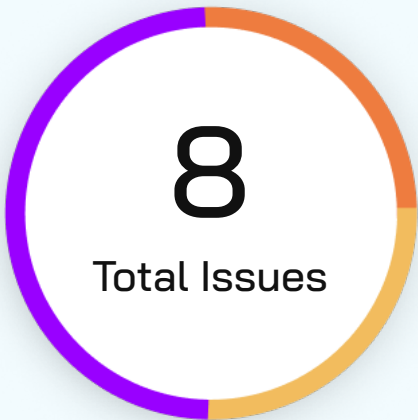


**Fixed In**

662ed82714d11d1efa5129ddac6b14ab0b4f9146

**Verify the Authenticity of Report on QuillAudits Leaderboard:**<https://www.quillaudits.com/leaderboard>

# Number of Issues per Severity



Critical	0 (0%)
High	0 (0%)
Medium	2 (25.0%)
Low	2 (25.0%)
Informational	4 (50.0%)

		Severity				
		Critical	High	Medium	Low	Informational
Issues	Open	0	0	0	0	0
	Acknowledged	0	0	1	2	2
	Partially Resolved	0	0	0	0	0
	Resolved	0	0	1	0	2



# Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Restake During Cooldown Leads to Stuck Withdrawal	Medium	Resolved
2	Merkle Leaf Construction Allows Cross-Round Replay Attacks	Medium	Acknowledged
3	Users Do Not Earn Rewards During Cooldown Period	Low	Acknowledged
4	Missing Validation for Total Percentage Allocation	Low	Acknowledged
5	Floating Solidity Pragma Used	Informational	Resolved
6	Ownable Should Use Two-Step Ownership Transfer	Informational	Acknowledged
7	Confusing Naming of Withdrawal Functions	Informational	Acknowledged
8	Unused EmergencyWithdrawal Event Declaration	Informational	Resolved



# 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

		Impact		
		High	Medium	Low
Likelihood	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.



# Medium Severity Issues

## Restake During Cooldown Leads to Stuck Withdrawal

**Resolved**

### Path

Staking.sol

### Path

```
restakeRewards(uint256 stakeId), withdraw(uint256 _stakeId)
```

### Description

The issue arises when a withdrawal has already been initiated for a given stake. At this point, the stake is marked as "in cooldown" and the corresponding principal amount is transferred into the pendingWithdrawals mapping while the stake entry itself is cleared (i.e., stake.amount = 0). This is intended to freeze the stake until the cooldown period expires. However, the restakeRewardsfunction does not verify whether a withdrawal is already pending for the same stakeId. As a result, users are able to call restakeRewards(stakeId) during the cooldown period.

This creates an inconsistent state: the contract allows rewards to be added back to the stake struct, but that stake is no longer valid because the associated funds are already in pendingWithdrawals. At the end of the cooldown, when finalizeWithdrawal is executed, the logic only transfers back the principal stored in pendingWithdrawals and completely ignores the newly restaked rewards. This means the rewards remain in the contract without any accessible path for the user to withdraw them.

### Impact

This leads to a situation where the user's principal is always safe and will be returned after cooldown, but any rewards restaked during the cooldown period are permanently stuck in the contract. From the user's perspective, this results in confusion and potential financial loss, since their balance will not match expectations after finalizing the withdrawal. If repeated, it could accumulate stranded funds inside the contract and reduce user trust in the staking system.

### Likelihood

Medium – While the scenario requires a specific sequence of actions, restaking is a normal behavior and a user may attempt it without realizing their stake is in cooldown. This makes the issue realistically exploitable.



## POC

```
1 function test_RestakeDuringCooldown_RewardsLost() public {
2     uint256 initialStake = 1 ether;
3
4     // 1. Stake
5     vm.startPrank(user1);
6     staking.createStake(initialStake);
7     vm.stopPrank();
8
9     console.log("User staked:", initialStake);
10
11    // 2. Accrue rewards
12    vm.warp(block.timestamp + 30 days);
13    vm.prank(owner);
14    staking.addRewards(1 ether, initialStake * 1 days, block.timestamp + 1 days);
15
16    uint256 pendingBefore = staking.getPendingRewards(user1, 0);
17    console.log("Pending rewards before withdraw:", pendingBefore);
18
19    // 3. Withdraw (starts cooldown)
20    vm.prank(user1);
21    staking.withdraw(0);
22    console.log("After withdraw: stake.amount = 0, pendingWithdrawal.amount = 1 ETH");
23
24    // 4. Restake rewards DURING cooldown
25    vm.prank(user1);
26    staking.restakeRewards(0);
27
28    console.log("User restaked rewards DURING cooldown");
29
30    (Staking.Stake[] memory stakes, uint256[] memory pending) = staking.getUserStakes(user1);
31    Staking.Stake memory s = stakes[0];
32    console.log("Stake amount after restake:", s.amount);
33
34    staking.withdraw(0);
35
36 }
37
38
```

## Recommendation

Redesign the withdrawal flow so that finalizeWithdrawal accounts for both principal and any restaked rewards, ensuring no tokens remain stuck.



## Merkle Leaf Construction Allows Cross-Round Replay Attacks

**Acknowledged**

### Path

WACHAirdropDistributor.sol

### Path

`claimAirdrop()`

### Description

The Merkle leaf construction in `claimAirdrop` includes `_index`, `msg.sender`, `round.startTime`, `round.endTime`, and `_percentageShare`, but not the `_roundId` itself:

```
1 // Verify Merkle proof
2 bytes32 leaf = keccak256(abi.encodePacked(_index, msg.sender, round.startTime, round.endTime, _percentageShare));
3 if (!MerkleProof.verify(_merkleProof, round.merkleRoot, leaf)) revert InvalidMerkleProof();
4
```

If two rounds happen to share the same `startTime` and `endTime` (for example, standardized monthly rounds), a valid Merkle proof from one round could potentially be replayed in another round if the same user appears in both with the same index and percentage. This opens the possibility of unauthorized double-claims across rounds.

### Impact

Users may be able to reuse a proof from one round to claim in another, leading to double-claiming and loss of tokens from the airdrop pool.

### Likelihood

The conditions (identical start/end times and matching index/percentage) are specific, but standardized round durations make it realistically possible.

### Recommendation

Include `_roundId` in the Merkle leaf construction to enforce round-level uniqueness

### WachAi Team's Comment

Every month, we airdrop \$WACH tokens, each with a different start date.





# Low Severity Issues

## Users Do Not Earn Rewards During Cooldown Period

**Acknowledged**

### Path

Staking.sol

### Path

```
withdraw(uint256 stakeId) finalize, Withdrawal(uint256 stakeId)
```

### Description

When a user initiates a withdrawal, the contract moves their principal into the cooldown state by transferring it to pendingWithdrawals and setting stake.amount = 0. However, the contract does not call updateReward at this point, meaning the user's rewards are not fully settled before cooldown starts. Since the stake is marked as zero, no further rewards accrue during the cooldown period. Effectively, the user loses any rewards they should have earned between the withdrawal request and the cooldown finalization.

This breaks the expected staking logic, because users typically assume they will continue earning rewards until their funds are actually withdrawn from the system. Instead, the rewards stop prematurely at the start of cooldown, leaving a gap where the user's capital is still locked but unproductive.

### Impact

Users do not receive the rewards for the entire cooldown period, leading to a direct financial loss and misalignment with user expectations.

### Likelihood

Every user who initiates a withdrawal is affected, since cooldown is a mandatory step.

### Recommendation

The design should account for this by extending reward calculation until finalizeWithdrawal

### WachAi Team's Comment

because we are not allowing that



## Missing Validation for Total Percentage Allocation

**Acknowledged**

### Path

WACHAirdropDistributor.sol

### Path

`createAirdropRound (...)`

### Description

The contract enforces that each individual claim has `_percentageShare <= MAX_PERCENTAGE`. However, there is no validation that the sum of all percentage shares in the Merkle tree equals 100% (or at least does not exceed 100%).

### Impact

Many users forget or are unaware of claim deadlines.

### Likelihood

Low

### Recommendation

Store and validate total allocation percentage in round metadata, ensuring it does not exceed 100%.

### WachAi Team's Comment

This will significantly increase the gas cost. We are performing this calculation on the backend



# Informational Issues

## Floating Solidity Pragma Used

**Resolved**

### Description

The contract uses a floating pragma `^0.8.20`. This allows compilation with any future 0.8.x version, which may introduce breaking changes or unintended behavior. Pinning the pragma to a fixed version ensures deterministic compilation and prevents unexpected behavior from newer compiler versions.

### Recommendation

Use a fixed pragma version instead of a floating one, e.g., `pragma solidity 0.8.20;`

## Ownable Should Use Two-Step Ownership Transfer

**Acknowledged**

### Description

The contract inherits from Ownable, which uses a single-step ownership transfer mechanism. This allows the current owner to directly set a new owner, and if the wrong address is passed, ownership could be lost permanently. A safer approach is to use a two-step ownership transfer, where the new owner must explicitly accept ownership.

### Recommendation

Use `TwoStepOwnable` from OpenZeppelin instead of `Ownable` to ensure safe ownership transfers. This requires the new owner to call `acceptOwnership`, reducing the risk of accidental misconfiguration.



## Confusing Naming of Withdrawal Functions

**Acknowledged**

### Path

Staking.sol

### Path

`withdraw`, `finalizeWithdrawal`, `undoWithdrawal`

### Description

The contract uses three different functions (`withdraw`, `finalizeWithdrawal`, and `undoWithdrawal`) to manage the withdrawal lifecycle. However, the naming is not intuitive:

`withdraw` does not actually withdraw tokens but only starts the cooldown (more like a “request”).

`finalizeWithdrawal` is the actual withdrawal action.

`undoWithdrawal` cancels a pending withdrawal.

This can confuse both developers and integrators, and may lead to mistakes in integration or user-facing interfaces. Clear naming is especially important in staking/withdrawal flows where funds are locked.

### Recommendation

Rename functions to match their actual behavior for better clarity:

`withdraw` → `requestWithdrawal` (initiates cooldown)

`finalizeWithdrawal` → `withdraw` (completes withdrawal)

`undoWithdrawal` → `cancelWithdrawal` (reverts cooldown request).

## Unused EmergencyWithdrawal Event Declaration

**Resolved**

### Path

WACHAirdropDistributor.sol

### Description

The contract declares an `EmergencyWithdrawal` event but never emits it. This suggests either an incomplete emergency withdrawal mechanism or leftover code.

### Recommendation

Either implement a proper `emergencyWithdraw` function that emits this event or remove the unused event.



# Functional Tests

Some of the tests performed are mentioned below:

- ✓ Tested staking, restaking, and withdrawing with cooldown under normal and edge conditions.
- ✓ Verified reward distribution across multiple stakes with overlapping timeframes.
- ✓ Checked behavior when addRewards is called with no active stakes.
- ✓ Validated finalizeWithdrawal and undoWithdrawal under different cooldown states.
- ✓ Tested Merkle airdrop claims with valid, invalid, and replayed proofs.
- ✓ Ensured IPFS hash updates work only for valid rounds.
- ✓ Verified percentage share limits and over-allocation scenarios.
- ✓ Checked contract pause/resume functionality and role-based access control.

# 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
Staking.sol	createstake	Checked for reentrancy, double-staking, zero amount deposits, and reward accounting accuracy.
	withdraw	Verified cooldown enforcement, reward settlement before withdrawal, and protection against premature/finalized withdrawals.
	finalizeWithdrawal	Ensured only valid pending withdrawals succeed, tested for double-finalization and missing reward accounting..
	undoWithdrawal	Validated proper restoration of stake, prevention of misuse after finalize.
WACHAirdropDistributor.sol	claimAirdrop	Tested valid vs invalid proofs, replay across rounds, double-claim prevention, and percentage share enforcement
	createAirdropRound	Checked max percentage limits, missing total allocation validation, and round initialization correctness.



# Closing Summary

In this report, we have considered the security of WachAi. We performed our audit according to the procedure described above.

Two Medium, Two low and Rest Acknowledged. Few of the issues were resolved and others were acknowledged by team

# 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.

While we have conducted a thorough review, security is an ongoing process. We strongly 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.

**7+**

Years of Expertise

**1M+**

Lines of Code Audited

**50+**

Chains Supported

**1400+**

Projects Secured

Follow Our Journey





# AUDIT REPORT

---

September 2025

For



**WachAI**



Canada, India, Singapore, UAE, UK

[www.quillaudits.com](http://www.quillaudits.com)

[audits@quillaudits.com](mailto:audits@quillaudits.com)