



AUDIT REPORT

July 2025

For

WEMO

Table of Content




Executive Summary	04
Number of Security Issues per Severity	06
Summary of Issues	07
Checked Vulnerabilities	08
Techniques and Methods	10
Types of Severity	12
Types of Issues	13
Severity Matrix	14
 High Severity Issues	15
1. Double Reward Claiming on Deposit via autoClaimForUsers()	15
 Medium Severity Issues	16
2. Inconsistent rebase() Handling Between Claim and AutoClaim	16
3. Missing stopInEmergency modifier in swap()	17
4. Inaccurate Balance Updates on Failed Swap in isAdditionalSwap Logic	18
 Low Severity Issues	20
5. No Function to Remove Pools	20
6. approve() incompatibility with some ERC20 tokens	21
 Informational Issues	22
7. Misleading Return Value in toggle()	22
Functional Tests	23
Threat Model	24



Table of Content

Automated Tests	33
Closing Summary & Disclaimer	33



Executive Summary

Project Name	Womofi
Protocol Type	Rebasing Token
Project URL	https://womo.finance/
Overview	<p>WOMO is a deflationary token protocol that implements an aggressive rebase mechanism to exponentially reduce total supply from 100M to 1M tokens over a 3-month period through epoch-based scaling. The protocol features a comprehensive ecosystem including bonding mechanics through SafeHaven for LP token deposits with vesting rewards, automated liquidity management via Treasury, staking rewards, and zipper functionality for streamlined user interactions. At its core, the WOMO token uses a global scaling factor applied to internal fragment balances, allowing all circulating supply to deflate proportionally without distinguishing between different holder types, creating a unique deflationary tokenomics model designed to compress value over time.</p>
Audit Scope	The scope of this Audit was to analyze the Womofi Smart Contracts for quality, security, and correctness
Source Code link	github.com/womofi/audit
Branch	audit
Contracts in Scope	Rebaser.sol, SafeHaven.sol, Staking.sol, Womo.sol, WomoTreasury.sol, ZapIn.sol
Commit Hash	361342ad8f5310fd864e648bd2011eb4bac16715
Language	Solidity
Blockchain	Sonic
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	26th June 2025 - 16th July 2025



Updated Code Received	16th July 2025
Review 2	16th July 2025
Fixed In	694e495cdb0290d33042cdc8a477393fefdc4200

Verify the Authenticity of Report on QuillAudits Leaderboard:

<https://www.quillaudits.com/leaderboard>



Number of Issues per Severity



Critical	0 (0%)
High	1 (14.2%)
Medium	3 (42.8%)
Low	2 (28.5%)
Informational	1 (14.2%)

		Severity				
		Critical	High	Medium	Low	Informational
Issues	Open	0	0	0	0	0
	Acknowledged	0	0	0	0	0
	Partially Resolved	0	0	0	0	0
	Resolved	0	1	3	2	1



Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Double Reward Claiming on Deposit via autoClaimForUsers()	High	Resolved
2	Inconsistent rebase() Handling Between Claim and AutoClaim	Medium	Resolved
3	Missing stopInEmergency modifier in swap()	Medium	Resolved
4	Inaccurate Balance Updates on Failed Swap in isAdditionalSwap Logic	Medium	Resolved
5	No Function to Remove Pools	Low	Resolved
6	Approve() incompatibility with some ERC20 tokens	Low	Resolved
7	Misleading Return Value in toggle()	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.



High Severity Issues

Double Reward Claiming on Deposit via `autoClaimForUsers()`

Resolved

Path

Staking.sol

Function

`deposit()`

Description

In the Staking.sol: `deposit()` function, the staking contract first calls `queueRewards()` for the depositing user before transferring LP tokens. However, it subsequently calls `autoClaimForUsers(_pid)`, which iterates over users (including the depositor, if in range) and invokes `_autoclaim()`. This second call to `_autoclaim()` also invokes `queueRewards()` before `user.rewardDebt` is updated, allowing the same pending reward to be added twice: Once during the initial `queueRewards()` call.

Again in `_autoclaim()` if the user is within `lastProcessedIndex` range.

This leads to over-rewarding and unbacked token minting, which can be exploited if deposits are carefully timed to fall within the auto-claim range.

Impact

Users can receive inflated rewards for the same staking position. Results in permanent inflation of the token supply `womo.mint()`

Recommendation

In `deposit()`, update `user.rewardDebt` immediately after `queueRewards()` and before calling `autoClaimForUsers()`, or exclude the depositing user from being auto-claimed in the current transaction.



Medium Severity Issues

Inconsistent rebase() Handling Between Claim and AutoClaim

Resolved

Description

`_claim()` : used for manual claims

`autoClaimForUsers()` : used during automated claiming, such as via direct or within `deposit()`. However, only `_claim()` calls `rebase.rebase()` before minting rewards with `womo.mint()`. The `auto ClaimForUsers()` function, which in turn calls `_autoclaim()`, skips this step, resulting in potential inconsistencies due to the mutable `tokensScalingFactor` used internally by `womo`.

Suppose:

A deflationary `rebase()` transaction occurring (e.g., `tokensScalingFactor` decreased)

`_autoclaim()` skips the rebase, using the old higher scaling factor, so user gets more tokens than they should

`_claim()` would rebase first, so it mints rewards using the new lower scaling factor (user gets fewer tokens)

Now two users claiming at the same time through different mechanisms get different real rewards.

Impact

Minting based on an outdated scaling factor. Unfair reward distribution between manual and auto claims. Reward accounting drift and inflation deviation over time

Recommendation

Ensure consistent behavior in both reward paths and out of gas operations should be taken into consideration.



Missing stopInEmergency modifier in swap()

Resolved

Path

ZapIn.sol

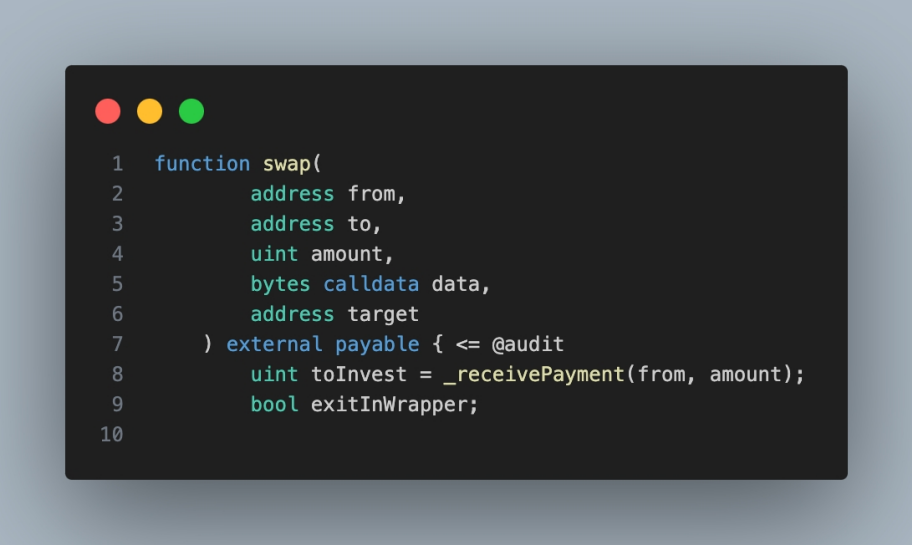
Function

`swap ()`

Description

The contract includes an emergency pause mechanism implemented via the `stopInEmergency` modifier, which prevents function execution when the `stopped` flag is set to true. This modifier is correctly applied to several critical functions such as `singleBond`, `duoBond`, and `singleAddStake`.

However, the `swap ()` function does not include the `stopInEmergency` modifier, which allows it to remain callable even when the contract is paused:



```
1  function swap(  
2      address from,  
3      address to,  
4      uint amount,  
5      bytes calldata data,  
6      address target  
7  ) external payable { <= @audit  
8      uint toInvest = _receivePayment(from, amount);  
9      bool exitInWrapper;  
10 }
```

Impact

When the contract is paused, `swap ()` can still be executed.

Recommendation

Apply the `stopInEmergency` modifier to the `swap ()` function

Inaccurate Balance Updates on Failed Swap in `isAdditionalSwap` Logic

Resolved

Path

ZapIn.sol

Function

`singleBond()`

Description

In the `isAdditionalSwap` branch, the contract attempts to rebalance token amounts by performing an internal swap using `_swapIntermediate()`, which relies on `_swapV2()` to execute a token swap via the router.

However, if the swap fails inside `_swapV2()` (e.g., due to slippage, routing failure, or an invalid path), the function does not revert, but instead returns zero as the bought amount:

```
1  uint amountOut;
2  try
3      ROUTER.swapExactTokensForTokens(
4          _amountToTrade,
5          1,
6          route,
7          address(this),
8          deadline
9      )
10     returns (uint[] memory amountsOut) {
11         amountOut = amountsOut[1];
12     } catch {
13         amountOut = 0; // @note
14     }
```

Despite the failure, the calling function proceeds to update `amountA` and `amountB` using the `amountSpent` and `amountBought` values:

```
1  if (remainingTokenA > remainingTokenB) {
2      (uint256 amountSpent, uint256 amountBought) = _swapIntermediate(
3          reserve0,
4          remainingTokenA,
5          tokenA,
6          tokenB
7      );
8
9      amountA -= amountSpent; // @audit modifies balance even if swap failed
10     amountB += amountBought;
```



Impact

May introduce subtle accounting drift across protocol interactions, resulting in inaccurate accounting

Recommendation

Ensure that balance updates only occur if the swap succeeds. This can be done by explicitly checking that `amountBought > 0` before modifying any values



Low Severity Issues

No Function to Remove Pools

Resolved

Path

Staking.sol

Description

The `add()` function allows the owner to add new staking pools by pushing to the `poolInfo` array. However, there is no corresponding function to remove a pool, which may lead to unnecessary accumulation of inactive or deprecated pools over time.



approve() incompatibility with some ERC20 tokens**Resolved****Path**

ZapIn.sol

Description

The contract uses `IERC20.approve` to interact with input tokens without using forceApprove or approving to zero first. However, there are some tokens that revert on such. This could lead to transaction reverts when interacting with the contract, or depositing tokens.



Informational Issues

Misleading Return Value in toggle()

Resolved**Path**

WomoTreasury.sol

Function Name

`toggle()`

Description

The **toggle()** function always returns true, regardless of whether the address was enabled or disabled. This may mislead integrators expecting the return value to indicate the new state.



Functional Tests

Some of the tests performed are mentioned below:

- ✓ rebase function executes correctly when called by approved caller after FRACTION_FACTOR time has passed
- ✓ rebase function syncs all registered UniswapV2 pairs after successful rebase
- ✓ rebase function handles multiple epoch differences correctly when time gap is large
- ✓ calculateDynamicCompoundRatio returns correct ratio to reach finalSupply over remaining rebases
- ✓ deposit function calculates fragments with additionalBips bonus correctly
- ✓ redeem function allows bond owner to redeem fully vested bonds



Threat Model

Contract	Function	Threats
Rebaser.sol -	<code>rebase()</code>	<p>Core function that executes the deflationary rebase mechanism, updating the global scaling factor and triggering supply compression.</p> <p>Inputs</p> <ul style="list-style-type: none"> - No direct inputs - Function reads from contract state - Control: Only approved callers via <code>onlyApprovedCallers</code> modifier - Constraints: Must be called by addresses in <code>approvedCallers</code> mapping - Impact: Triggers global supply deflation affecting all token holders <p>Branches and Code Coverage</p> <p>Intended Branches</p> <ul style="list-style-type: none"> - Should execute rebase only when <code>lastTime + FRACTION_FACTOR <= block.timestamp</code> - Should stop rebasing when <code>womo.totalSupply() < finalSupply</code> - Should calculate correct compound ratio using <code>_calculateDynamicCompoundRatio()</code> when <code>computedOnTheFly</code> is true - Should sync all registered UniswapV2 pairs after rebase



Contract	Function	Threats
		<ul style="list-style-type: none"> - Should update staking rewards proportionally to deflated supply when <code>useAutoStakingAdjustment</code> is true - Should adjust <code>compoundRatio</code> via <code>adjust()</code> function after each rebase <p>Negative Behavior</p> <ul style="list-style-type: none"> - Should not allow unauthorized callers to trigger rebase - Should not rebase more frequently than <code>FRACTION_FACTOR</code> interval - Should not allow rebase when <code>rebaseCount</code> is zero and <code>computedOnTheFly</code> is true <p>Security Concerns</p> <ul style="list-style-type: none"> - Front-running: MEV bots can sandwich rebase calls - Precision loss: Compound calculations may lose precision over time - Governance risk: Malicious approved caller can drain protocol
SafeHaven.sol	<code>deposit()</code>	<p>Allows users to deposit LP tokens and receive WOMO tokens with bonus after vesting period.</p> <p>Inputs</p> <p><code>_to</code></p> <ul style="list-style-type: none"> Control: Fully controlled by caller Constraints: Must not be address(0) Impact: Address that will receive the bond



Contract	Function	Threats
		<p>amountDesiredA, amountDesiredB Control: Fully controlled by caller Constraints: Passed to Treasury contract for validation Impact: Amounts of tokens to deposit for LP creation</p> <p>amountAMin, amountBMin Control: Fully controlled by caller Constraints: Slippage protection parameters Impact: Minimum amounts for LP creation</p> <p>deadline Control: Fully controlled by caller Constraints: Must be future timestamp Impact: Transaction deadline for LP creation</p> <h2>Branches and Code Coverage</h2> <h3>Intended Branches</h3> <ul style="list-style-type: none"> - Should create LP tokens via Treasury contract - Should calculate fragments with <code>additionalBips</code> bonus - Should reserve correct amount of tokens using <code>_debtCalculator()</code> - Should create new Bond struct with correct vesting parameters - Should add user to active bonds tracking if first bond - Should trigger rebase after deposit



Contract	Function	Threats
		<ul style="list-style-type: none"> - Should process expired bonds automatically <p>Negative Behavior</p> <ul style="list-style-type: none"> - Should not allow deposit to zero address - Should not allow deposit when reserves are insufficient - Should not allow deposit when maxSupply would be exceeded - Should not allow deposit with invalid LP parameters <p>Security Concerns</p> <ul style="list-style-type: none"> - Debt calculation vulnerability: <code>_debtCalculator()</code> assumes future rebase behavior - Reserve exhaustion: No global cap on total bonds - Reentrancy: External calls to Treasury and Rebaser
SafeHaven.sol	<code>redeem()</code>	<p>Allows users to claim their vested WOMO tokens from bonds.</p> <p>Inputs</p> <p><code>_recipient</code></p> <ul style="list-style-type: none"> Control: Fully controlled by caller Constraints: Must have active bond at bid index Impact: Address to receive redeemed tokens



Contract	Function	Threats
		<p>bid</p> <p>Control: Fully controlled by caller</p> <p>Constraints: Must have active bond at bid index</p> <p>Impact: Address to receive redeemed tokens</p> <p>Branches and Code Coverage</p> <p>Intended Branches</p> <ul style="list-style-type: none"> - Should only allow recipient or fully vested bonds to be redeemed - Should remove bond from array when fully vested - Should mint correct amount of tokens based on fragments and reserves - Should handle cases where fragments exceed reserved tokens - Should remove user from active list when no bonds remain - Should trigger rebase after redemption <p>Negative Behavior</p> <ul style="list-style-type: none"> - Should not allow redemption of non-existent bonds - Should not allow unauthorized redemption of others' bonds - Should not allow redemption of unvested bonds by non-owners - Should not allow double redemption of same bond

Contract	Function	Threats
		<p>Security Concerns</p> <ul style="list-style-type: none"> - Reward calculation: Complex reward math could have precision issues - Lock mechanism: Users can't withdraw until lock expires - Auto-claim: Gas griefing through excessive auto-claims
Staking.sol	<code>withdraw()</code>	<p>Allows users to withdraw their staked tokens after lock period expires.</p> <p>Inputs</p> <p><u><code>_pid</code></u> Control: Fully controlled by caller Constraints: Must be valid pool ID Impact: Determines which pool to withdraw from</p> <p><u><code>_amount</code></u> Control: Fully controlled by caller Constraints: Must be > 0 and \leq user's staked amount Impact: Amount of tokens to withdraw</p> <p>Branches and Code Coverage</p> <p>Intended Branches</p> <ul style="list-style-type: none"> - Should only allow withdrawal after lock period expires - Should update pool rewards before processing withdrawal

Contract	Function	Threats
		<ul style="list-style-type: none"> - Should queue and claim pending rewards - Should handle WOMO token withdrawals with fragment conversion - Should update user's amount and reward debt - Should remove user from pool tracking if full withdrawal - Should trigger auto-claim for other users <p>Negative Behavior</p> <ul style="list-style-type: none"> - Should not allow withdrawal during lock period - Should not allow withdrawal of more than staked amount - Should not allow zero amount withdrawals - Should not allow withdrawal from non-existent pools <p>Security Concerns</p> <ul style="list-style-type: none"> - Lock bypass: Must ensure lock period cannot be circumvented - Reward manipulation: Withdraw timing could be gamed for rewards - State consistency: User removal logic must be bulletproof



Contract	Function	Threats
Rebaser.sol	<code>_calculateDynamicCompoundRatio()</code>	<p>Calculates the compound ratio needed to reach final supply target over remaining rebase periods.</p> <p>Inputs</p> <ul style="list-style-type: none"> - No direct inputs - Reads from contract state - Control: Internal function called by <code>rebase()</code> - Constraints: Requires <code>circulatingSupply > finalSupply</code> and <code>rebaseCount > 0</code> - Impact: Determines the rate of supply deflation <p>Branches and Code Coverage</p> <p>Intended Branches</p> <ul style="list-style-type: none"> - Should calculate correct decay factor using logarithmic math - Should ensure supply converges to <code>finalSupply</code> over <code>rebaseCount</code> periods - Should handle edge cases when very close to final supply - Should revert when already at or below target supply - Should revert when no rebases are left <p>Negative Behavior</p> <ul style="list-style-type: none"> - Should not allow calculation when <code>supply <= finalSupply</code> <p>Should not allow calculation when <code>rebaseCount = 0</code></p>



Contract	Function	Threats
		<ul style="list-style-type: none">- Should not produce ratio $> 1e18$ (100%)- Should not cause arithmetic overflow/underflow Security Concerns <ul style="list-style-type: none">- Mathematical precision: ABDK library precision limitations- Edge cases: Behavior near final supply or zero rebases- Convergence: May not reach exact final supply due to rounding



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.

Closing Summary

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

Issues of High, Medium and Low severity were found. Womofi team resolved all the issues mentioned

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

July 2025

For

WEMO



Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillaudits.com