QuillAudits

AUDIT REPORT

January 2026

For

alvara

Table of Content

Executive Summary

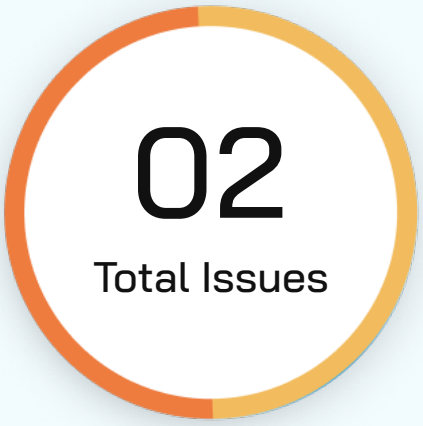
Project Name	Alvara
Protocol Type	Merkle-based reward distribution
Project URL	https://alvara.xyz
Overview	The GaugeWeightClaims contract implements a Merkle-based reward distribution mechanism for allocating ALVA tokens to users based on off-chain gauge weight voting results. For each proposal or epoch, the contract owner publishes a Merkle root representing eligible accounts and their reward amounts, which users can then claim by submitting a valid Merkle proof. The contract tracks claimed entries using a bitmap to prevent double-claims, supports both single and batch claim execution, and allows root versioning to manage updates prior to claims. Overall, the design relies on off-chain computation and owner-managed root publication, making correctness and fairness dependent on the integrity of the Merkle tree generation and administrative controls.
Audit Scope	The scope of this Audit was to analyze the Alvara Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Alvara-Protocol/AlvaraDAO
Contracts in Scope	gaugeweightclaims.sol
Commit Hash	8544943796af1e72e249fe395639dd12ca414d74
Branch	claim-portal
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	23rd December 2025 to 24th December 2025
Updated Code Received	5th January 2026
Review 2	5th January 2026

Fixed In	https://github.com/Alvara-Protocol/AlvaraDAO/commit/56fd8d72b145845d9e8f2e5bd8d17360216d1d27

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0.0%)
High	0 (0.0%)
Medium	1 (50.0%)
Low	1 (50.0%)
Informational	0 (0.0%)

Severity

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

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Anybody can claim for anyone	Medium	Resolved
2	Consider using Ownable2Step	Low	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
- 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

Likelihood	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.

Medium Severity Issues

Anybody can claim for anyone

Resolved

Path

gaugeweightclaims.sol

Function Name

`claim()`

Description

The claim and batchClaim functions allow any caller to execute a claim on behalf of any recipient, as the account parameter is user-supplied and not validated against msg.sender.

As a result, any party observing the mempool can copy a pending claim transaction—including its Merkle proof—and submit the same claim with a higher gas price. Since claim uniqueness is enforced solely by the (proposalId, index) pair, the first transaction to be mined succeeds, while subsequent attempts revert with Already claimed.

Although the reward tokens are transferred to the intended recipient, this behavior enables third parties to preemptively execute claims for other users.

```
function _claim(
        uint256 proposalId,
        uint256 index,
        address account, //@audit anybody can claim for anyone, should be msg.sender
        uint256 amount,
        bytes32[] calldata merkleProof
    ) internal {
        require(account != address(0), "Zero account");

        bytes32 root = merkleRoots[proposalId];
        require(root != bytes32(0), "Root not set");
        require(!_isClaimed(proposalId, index), "Already claimed");

        bytes32 node = keccak256(
            abi.encodePacked(proposalId, index, account, amount)
        );

        require(MerkleProof.verify(merkleProof, root, node), "Invalid proof");

        _setClaimed(proposalId, index);

        require(ALVA.transfer(account, amount), "ALVA transfer failed");

        emit Claimed(proposalId, index, account, amount);
    }

```

Impact

This issue does not allow theft or redirection of funds, as rewards are always transferred to the address encoded in the Merkle leaf. However, it enables the following negative outcomes:

Griefing / denial of execution: Legitimate users may have their claim transactions reverted after being front-run.

Likelihood

Medium

The attack requires no special privileges and can be executed by any observer monitoring the public mempool. Since Merkle proofs are submitted in plaintext, front-running is straightforward and inexpensive.

POC

1. User submits a transaction calling claim(...) with valid parameters and Merkle proof.
2. An attacker observes the pending transaction in the mempool
3. The attacker copies the exact calldata and resubmits it with a higher gas price.
4. The attacker's transaction is mined first, successfully executing the claim.
5. The original user's transaction reverts with Already claimed.

Recommendation

If permissionless claiming is not intended, restrict claim execution to the reward recipient

Low Severity Issues

Consider using Ownable2Step

Resolved

Path

gaugeweightclaims.sol

Description

The project intends to use Ownable2Step semantics (safe 2-step ownership transfer), but currently inherits from Ownable, not Ownable2Step.

This means the owner can instantly transfer ownership in one transaction defeating the intended Two-Step security model documented in comments and project requirements.

If the owner key is compromised or misused, privileged control can be reassigned instantly without the beneficiary's acceptance.

Impact

Low

Likelihood

Low

Recommendation

Replace Ownable with Ownable2Step to enforce a safer two-step ownership transfer process.

Functional Tests

- Allows claim execution by a non-recipient address
- Transfers rewards to the encoded recipient, not the caller
- Prevents the original user from executing the claim after preemption
- Prevents double-claiming for the same proposal and index

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
gaugeweightclaims.sol	setMerkleRoot(uint256 proposalId, bytes32 merkleRoot, uint256 version)	Allows the owner to set or update the Merkle root for a proposal. **Inputs** **proposalId** • Control: Fully controlled by the owner. • Constraints: Must be non-zero. • Impact: Identifies the proposal for which rewards become claimable. **merkleRoot** • Control: Fully controlled by the owner. • Constraints: Must be non-zero. • Impact: Defines the entire reward distribution for the proposal. **version** • Control: Fully controlled by the owner. • Constraints: Must be greater than the current stored version. • Impact: Prevents accidental reuse of older Merkle roots. **Branches and code coverage** **Intended branches** • Should allow setting a Merkle root when no claims have been made. • Should reject Merkle roots with non-incrementing versions.

Contract	Function	Threats
		Test coverage • Should store the new Merkle root for the given proposal. • Should update the root version correctly. **Negative behavior** • Should not allow setting a zero Merkle root. • Should not allow setting a Merkle root with a stale version. • Should not allow non-owners to set the Merkle root.
	claim(uint256 proposalId, uint256 index, address account, uint256 amount, bytes32[] calldata merkleProof)	Allows a user to claim rewards for a proposal. **Inputs** **proposalId** • Control: Fully controlled by the user. • Constraints: Merkle root must be set. • Impact: Determines which proposal is being claimed. **index** • Control: Fully controlled by the user. • Constraints: Must not have been claimed previously. • Impact: Used to track claim uniqueness. **account** • Control: Fully controlled by the user. • Constraints: Must be non-zero. • Impact: Address that receives the reward tokens.

Contract	Function	Threats
		amount • Control: Fully controlled by the user. • Constraints: Must match the Merkle leaf. • Impact: Amount of tokens transferred. **merkleProof** • Control: Fully controlled by the user. • Constraints: Must be valid for the stored Merkle root. • Impact: Authorizes the claim. **Branches and code coverage** **Intended branches** • Should verify the Merkle proof successfully. • Should prevent double-claiming for the same index. • Should transfer tokens to the recipient account. **Test coverage** • Should mark the claim index as claimed. • Should transfer the correct amount of tokens to the account. **Negative behavior** • Should not allow claiming with an invalid Merkle proof. • Should not allow claiming the same index twice. • Should not allow claiming with a zero account address.

Contract	Function	Threats
	batchClaim(uint256[] proposalIds, uint256[] indices, address[] accounts, uint256[] amounts, bytes32[][] merkleProofs)	Allows batch claiming of rewards across multiple proposals. **Inputs** **proposalIds** • Control: Fully controlled by the user. • Constraints: Length must match • other input arrays. Impact: Identifies proposals to claim from. **indices** • Control: Fully controlled by the user. • Constraints: Length must match other input arrays. • Impact: Used to track claim uniqueness. **accounts** • Control: Fully controlled by the user. • Constraints: Length must match other input arrays. • Impact: Recipients of the rewards. **amounts** • Control: Fully controlled by the user. • Constraints: Length must match other input arrays. • Impact: Amounts to be transferred. **merkleProofs** • Control: Fully controlled by the user. • Constraints: Length must match other input arrays. • Impact: Proofs authorizing each claim.

Contract	Function	Threats
		Branches and code coverage Intended branches • Should execute all claims when all entries are valid. • Should revert the entire transaction if any claim fails. Test coverage • Should process multiple valid claims in a single transaction. **Negative behavior** • Should revert if input array lengths mismatch. • Should revert if any claim has already been processed.
	rescueTokens(address to, uint256 amount)	Allows the owner to withdraw ALVA tokens from the contract. **Inputs** to • Control: Fully controlled by the • owner. • Constraints: Must be non-zero. Impact: Recipient of the rescued tokens. amount • Control: Fully controlled by the owner. • Constraints: None. • Impact: Amount of tokens withdrawn.

Contract	Function	Threats
		Branches and code coverage Intended branches • Should transfer tokens to the specified address. Test coverage Should transfer the specified amount of tokens successfully. **Negative behavior** • Should not allow non-owners to rescue tokens. • Should not allow rescuing tokens to the zero address.

•

Centralization Risk

The above audit works under the assumption that :

- ALVA token is not malicious or shows non standard ERC20 behavior
- Owner role is trusted

It is worth noting that the contract has specialized owner priviledges via rescueToken where tokens can be transferred to any directed address.

Closing Summary

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

No critical issues in, just 2 issues of Medium and Low severity were found. Alvara team resolved the mentioned isssues

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

January 2026

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com