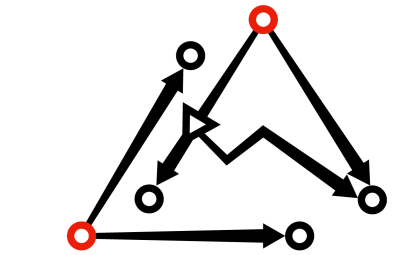

Least Authority

PRIVACY MATTERS

Airdrop Contract

Security Audit Report

Aligned Layer

Final Audit Report: 21 July 2026

Table of Contents

Overview

Background

Aligned Layer has requested that Least Authority perform a security audit of its `ClaimableAirdrop.sol` contract.

Project Dates

- **June 16, 2026 - June 17, 2026:** Initial Code Review *(Completed)*
- **June 17, 2026:** Delivery of Initial Audit Report *(Completed)*
- **Jul 20, 2026:** Verification Review*(Completed)*
- **Jul 21, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- Will Sklenars, Security Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the `ClaimableAirdrop.sol` contract followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository was considered in scope for the review:

- `ClaimableAirdrop.sol`: https://github.com/yetanotherco/aligned_layer/blob/testnet/claim_contracts/src/ClaimableAirdrop.sol

Specifically, we examined the Git revision for our initial review:

- `f5aed29e6a3c70b714dc06ad13bb98e26f63ebe5`

For the verification, we examined the following Git revision:

- `28a2a0be21879ddaa9c34770bbecb6cd1c0394a6`

For the review, this repository was cloned for use during the audit and for reference in this report:

- https://github.com/LeastAuthority/Aligned-Layer/tree/testnet

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://alignedlayer.com

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

`ClaimableAirdrop` enables Aligned Layer to distribute tokens to early users in a gas-efficient manner by requiring users to provide a Merkle proof when claiming tokens. This prevents tokens from being wasted on non-engaged users. We did not identify any path by which tokens can be stolen from the contract, any mechanism by which an unauthorized user can interfere with it, or any vulnerability to front-running. We examined the functionality and carefully considered edge cases, including the contract owner's ability to revoke approvals or otherwise control the contract in a manner that may not be expected by users.

System Design

`ClaimableAirdrop` is a straightforward airdrop contract in which the contract owner sets a Merkle root for a set of approvals consisting of `(claim owner, amount, and validFrom timestamp)` tuples. Each approval may be claimed by its owner by calling `claim` with the Merkle proof, amount, and `validFrom` timestamp that link the claim leaf to the current `claimMerkleRoot`. The only information available on-chain is the Merkle root, so the owner must also provide the proof, amount, and timestamp to the owners. An approval may not be claimed more than once, and only the full amount of an approval may be claimed. Claim owners may claim more than one approval at once using `claimBatch`, provided that the amount, `validFrom` timestamp, or both are unique for each claim. The contract owner can use the same contract for multiple airdrop rounds by calling `updateMerkleRoot`. The contract cannot perform validity checks on the leaves in the tree. Accordingly, an updated Merkle root may no longer contain leaves that have not yet been claimed. As a result, the contract owner has the implicit ability to revoke unused claims. Additionally, because the contract cannot validate whether the amount available to the `tokenDistributor` is adequate to cover all approved claims, it is implicitly possible to operate the

contract on a first-come, first-served basis. Claim owners should therefore exercise their claims as soon as possible.

For contract methods that can result in calls to external contracts (`claim`, `claimBatch`), OpenZeppelin's `nonReentrant` is used to mitigate contract reentrancy attacks.

Dependencies

Our team did not identify any dependency-related security concerns. The contract relies only on well-established OpenZeppelin contracts.

Code Quality

We performed a manual review of the repositories in scope and found the code to be well-organized and aligned with development best practices.

Tests

The in-scope contract does not include test coverage. We recommend implementing a test suite to help identify any implementation errors that could lead to security vulnerabilities (Suggestion 2).

Documentation and Code Comments

The available documentation is sufficient, accurate, and helpful. The code is also well commented.

Scope

The scope of this review was limited to the `ClaimableAirdrop.sol` contract. Related contracts, such as the `tokenDistributor`, are therefore assumed to operate as intended.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Suggestion 1: Document Edge Cases	Informational	Resolved
Suggestion 2: Implement a Test Suite	Informational	Resolved

Suggestions

Suggestion 1: Document Edge Cases

Location

`claim_contracts/src/ClaimableAirdrop.sol#L180-L184`

`claim_contracts/src/ClaimableAirdrop.sol#L135-L140`

Synopsis

The contract owner can revoke unclaimed approvals and operate the airdrop on a first-come, first-served basis. These behaviors should be documented.

Technical Details

The on-chain contract only updates the tree root hash, and `claim` only verifies that a given Merkle proof validates whether the specified leaf is part of the tree. Examining the entire tree is infeasible because it would require a large amount of calldata and high gas costs. Accordingly, certain system behavior cannot be enforced, so there are implicit edge cases that should be documented.

The contract owner can revoke unclaimed approvals by updating the root hash. Since the Merkle root can be updated, it can be replaced with a root hash that no longer contains previously valid but unclaimed approvals.

Transactions from the `tokenDistributor` may fail. Although the approval contains an amount, final transfer remains dependent on the `tokenDistributor`. However, a token distributor may not be sufficiently funded to service every approval. Consequently, the remaining claims may fail on a first-come, first-served basis.

Mitigation

We recommend documenting the above features clearly.

Status

The Aligned Layer team has updated the contract-level NatSpec to explicitly document that:

- The owner can revoke unclaimed approvals by replacing the Merkle root.
- Distribution becomes effectively first-come, first-served when the distributor lacks sufficient balance or allowance.

Verification

Resolved.

Suggestion 2: Implement a Test Suite

Synopsis

Our team found no tests for the in-scope contract. Sufficient test coverage of both success and failure paths can help illuminate edge cases that may develop into vulnerabilities.

Mitigation

We recommend creating a test suite to facilitate the detection of implementation errors and potential security vulnerabilities.

Status

The Aligned Layer team has implemented a Foundry test suite comprising 24 tests to cover the success and failure paths of `claim` and `claimBatch`, access-control restrictions, and ERC-20 interactions.

Verification

Resolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:
https://leastauthority.com/security-consulting.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.