OpenZeppelin | security

Aligned Token Audit

ALIGNED

December 23, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-12-15 To 2024-12-17
Languages	Solidity

Total Issues	19 (13 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	6 (3 resolved, 1 partially resolved)
Notes & Additional Information	13 (10 resolved)

Scope

We audited the yetanotherco/aligned_layer repository at commit b8c81088.

In scope were the following files:

```
claim_contracts
└── src
    ├── AlignedToken.sol
    └── ClaimableAirdrop.sol
```

System Overview

The scope includes the `AlignedToken` contract which is an ERC20-compliant token, and the `ClaimableAirdrop` contract which is designed to distribute a portion of these tokens via an airdrop.

`AlignedToken`

The `AlignedToken` contract is ERC20-compliant and implements permit functionality as defined in EIP-2612. It has an initial supply of 10 billion tokens and is upgradeable using the Transparent Proxy pattern. The contract also incorporates `Ownable2Step` functionality to manage permissioned operations. At the audited commit, the owner does not have any special privileges, but this role could potentially be used for burning, minting, or other operations in the future.

`ClaimableAirdrop`

The `ClaimableAirdrop` contract is responsible for distributing a portion of the `AlignedToken` supply. It includes logic that allows users to claim tokens from the airdrop, pause and unpause specific functionalities, and update configurable variables such as the claim period and the Merkle tree root. The contract also implements `Ownable2Step`. Like `AlignedToken`, this contract is also upgradeable through the Transparent Proxy pattern.

Merkle Tree

A set of addresses for all airdrop recipients, along with their respective claim amounts, is used to generate a Merkle Tree. The Merkle tree root is set during initialization. This setup enables users to claim their airdrop tokens by submitting a Merkle proof on-chain, which verifies their inclusion in the Merkle Tree. Each address can claim tokens only once and for the pre-defined amount specified in the Merkle Tree.

Privileged Roles

The following roles are assigned during the deployment of the contracts:

`AlignedToken`

- `_foundation`: An address that receives 7.3B of the token supply upon initialization. This address is also the owner of the `AlignedToken` contract.

- `_claimSupplier`: An address that receives 2.7B of the token supply, intended for the airdrop. This address must approve a token allowance for the airdrop contract to enable the distribution of tokens among airdrop recipients.

`ClaimableAirdrop`

The contract owner can pause and unpause the `claim`, `updateMerkleRoot`, and `extendClaimPeriod` functions. Notably, when the `claim` function is paused, the `updateMerkleRoot` and `extendClaimPeriod` functions are unpaused, and vice versa—when these functions are paused, the `claim` function is unpaused.

The owner can also extend the claim period, which must be greater than both the previous claim period and the current block timestamp. In addition, the owner can update the Merkle root using the `updateMerkleRoot` function. Regardless of the new root value and new associated proofs, users who have already claimed tokens cannot claim them again, as claims are tracked by account address.

Security Model and Trust Assumptions

The `ClaimableAirdrop` contract distributes tokens to various addresses based on a Merkle root. It is assumed that the Merkle root is generated correctly and that the information used to construct the tree is shared publicly, allowing users to submit their proofs on-chain.

It is also assumed that all privileged roles mentioned earlier are trustworthy and have been carefully selected by the Aligned team.

Low Severity

L-01 Total Supply Redundancy

The `AlignedToken` contract defines a `TOTAL_SUPPLY` constant as `10_000_000_000`. However, the `ERC20Upgradeable` contract already tracks the total supply with the `totalSupply` variable, which is accessible via the `totalSupply` function. Even though the minted amounts match the `TOTAL_SUPPLY`, the value is redundant and could become error-prone in the future if the `AlignedToken` contract is upgraded to include burn or mint functionality.

Consider removing the redundancy and relying on the variables provided by the `ERC20Upgradeable` contract instead.

Update: *Resolved in pull request #1623.*

L-02 Suggestions for `ReentrancyGuard` Usage

The `ClaimableAirdrop` contract uses the `ReentrancyGuard` contract to avoid reentrancy attacks in the `claim` function. However, since this contract is upgradeable, it should use the upgradeable version of the contract from the `@openzeppelin/contracts-upgradeable` repository.

In addition, if the airdrop contract is only deployed on Ethereum Mainnet, it is safe to use the `TransientReentrancyGuardUpgradeable` contract, which leverages transient storage to reduce gas costs. Finally, if there are no plans to add hooks or external calls in the `AlignedToken` contract in the future, and since the only external call currently performed is to the token contract itself, reentrancy protection can be removed entirely by following the checks-effects-interactions (CEI) pattern.

Consider implementing the following recommendations:

- Use `ReentrancyGuardUpgradeable` from the `@openzeppelin/contracts-upgradeable` repository for consistency with upgradeable contracts.
- Use `TransientReentrancyGuardUpgradeable` to reduce gas costs on Ethereum Mainnet.

- Remove reentrancy protection and follow the CEI pattern if future hooks or external calls are not planned.

Update: *Partially resolved in pull request #1636. While the upgradeable version of the `ReentrancyGuard` is now utilized by the contract, the other recommendations have not been implemented.*

L-03 Functions Are Updating the State Without Event Emissions

Within `ClaimableAirdrop.sol`, multiple instances of functions updating state without emitting events were identified:

- The `updateMerkleRoot` function
- The `extendClaimPeriod` function

To avoid hindering users from tracking off-chain activities or monitoring state changes, consider emitting events whenever the state is updated. This improves transparency, facilitates off-chain integrations, and enhances the overall readability and maintainability of the codebase.

Update: *Resolved in pull request #1636.*

L-04 Outdated Solidity Version and Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Throughout the codebase, multiple instances of floating and outdated pragma directives were identified:

- `AlignedToken.sol` has the solidity ^0.8.19 floating pragma directive.
- `ClaimableAirdrop.sol` has the solidity ^0.8.13 floating pragma directive.

Consider using fixed pragma directives and upgrading to the latest stable Solidity version.

Update: *Resolved in pull request #1623.*

L-05 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `AlignedToken.sol`, the `AlignedToken` contract
- In `ClaimableAirdrop.sol`, the `ClaimableAirdrop` contract

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not resolved.*

L-06 Incomplete Docstring

Within `ClaimableAirdrop.sol`, the `to` and `amount` parameters of the `TokenClaimed` event are not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not resolved.*

Notes & Additional Information

N-01 Merkle Root Update Could Lead to Undesired Outcomes

The `updateMerkleRoot` function allows the owner to update the Merkle tree root while claims are paused. If the root and proofs are updated and a user who has already claimed tokens attempts to claim a new token amount introduced in the updated tree, the claim will fail. This occurs because claims are tracked by addresses, and the function call will revert.

Consider documenting this behavior to ensure transparency and to set clear expectations for the users interacting with the system.

Update: *Acknowledged, not resolved. The team stated:*

> *Acknowledged. Root shouldn't be changed in production.*

N-02 Use Custom Errors

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed.

Throughout the codebase, multiple instances of `revert` and/or `require` messages were identified:

- In the `AlignedToken.sol` file, the `require` statement with the message "Invalid _foundation or _claimSupplier"
- In the `AlignedToken.sol` file, the `revert` statement with the message "Cannot renounce ownership"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid owner address"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid token contract address"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid token owner address"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid timestamp"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid Merkle root"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Account has already claimed the drop"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Drop is no longer claimable"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid Merkle proof"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Insufficient token allowance"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Failed to transfer funds"

- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Invalid root"
- In the `ClaimableAirdrop.sol` file, the `require` statement with the message "Can only extend from current timestamp"

For conciseness and gas savings, consider replacing `require` and `revert` messages with custom errors.

Update: *Acknowledged, not resolved. The team stated:*

> *Acknowledged. It's a good suggestion. Due to time constraints we didn't make this change.*

N-03 Magic Numbers in the Code

Within `AlignedToken.sol`, multiple instances of literal values with unexplained meaning were identified:

- The `7_300_000_000e18` literal number
- The `2_700_000_000e18` literal number

Consider defining and using `constant` variables instead of using literals to improve the readability of the codebase.

Update: *Resolved in pull request #1636.*

N-04 Redundant Modifiers

The `whenNotPaused` and `whenPaused` modifiers from the pause and unpause functions of the `ClaimableAirdrop` contract are already implemented in the `_pause` and `_unpause` functions of the `PausableUpgradeable` contract.

Consider removing the `whenNotPaused` and `whenPaused` modifiers from the pause and unpause functions of the `ClaimableAirdrop` contract.

Update: *Resolved in pull request #1636.*

N-05 Function Visibility Overly Permissive

Throughout the codebase, multiple instances of functions with unnecessarily permissive visibility were identified:

- The `initialize` function in `AlignedToken.sol` with `public` visibility could be limited to `external`.
- The `initialize` function in `ClaimableAirdrop.sol` with `public` visibility could be limited to `external`.
- The `claim` function in `ClaimableAirdrop.sol` with `public` visibility could be limited to `external`.
- The `pause` function in `ClaimableAirdrop.sol` with `public` visibility could be limited to `external`.
- The `unpause` function in `ClaimableAirdrop.sol` with `public` visibility could be limited to `external`.

To better convey the intended use of functions and to potentially realize some additional gas savings, consider changing a function's visibility to be only as permissive as required.

Update: *Resolved in pull request #1636.*

N-06 Multiple Optimizable State Reads

In the `ClaimableAirdrop` contract, the `tokenProxy` and `claimSupplier` variables are accessed from storage multiple times. This can lead to unnecessary gas costs in the following lines: 101, 106, and 107.

Consider defining the `claimSupplier` and `tokenProxy` variables as immutable and setting them in the constructor, as they cannot be updated in the current implementation. Alternatively, consider caching these values in memory variables to reduce storage access.

Update: *Acknowledged, not resolved. The team stated:*

> *Acknowledged. We left this gas optimization outside due to time constraints.*

N-07 Non-Explicit Imports

The use of non-explicit imports in the codebase can decrease code clarity and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity file or when inheritance chains are long.

Throughout the codebase, multiple instances of non-explicit imports were identified:

- The import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol"; import in `AlignedToken.sol`
- The import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol"; import in `AlignedToken.sol`
- The import "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC20PermitUpgradeable.sol"; import in `AlignedToken.sol`
- The import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol"; import in `AlignedToken.sol`
- The import "@openzeppelin/contracts/token/ERC20/IERC20.sol"; import in `ClaimableAirdrop.sol`
- The import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol"; import in `ClaimableAirdrop.sol`
- The import "@openzeppelin/contracts/utils/ReentrancyGuard.sol"; import in `ClaimableAirdrop.sol`
- The import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol"; import in `ClaimableAirdrop.sol`
- The import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol"; import in `ClaimableAirdrop.sol`
- The import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol"; import in `ClaimableAirdrop.sol`

Following the principle that clearer code is better code, consider using the named import syntax (*`import {A, B, C} from "X"`*) to explicitly declare which contracts are being imported.

Update: *Resolved in pull request #1636.*

N-08 Convoluted `Owner2StepUpgradeable` Initialization

In the `initialize` functions of the `AlignedToken` and `ClaimableAirdrop` contracts, the call to the `__Ownable2Step_init` function is unnecessary as this function is implemented as empty. Furthermore, instead of directly invoking the `_transferOwnership` function, the owner address can be passed as a parameter to the `__Ownable_init` function. This eliminates the need to manually check owner addresses, as `__Ownable_init` already ensures that the address is not zero.

Consider removing the redundant `__Ownable2Step_init` call and using `__Ownable_init` with the owner parameter to simplify the code and reduce gas costs.

Update: *Resolved in pull request #1636.*

N-09 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts missing a security contact were identified:

- The `AlignedToken` contract
- The `ClaimableAirdrop` contract

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #1649 and pull request #1639.*

N-10 Naming Suggestion

In the airdrop, more than one token can be claimed by users. As a result, the `TokenClaimed` event name is misleading since it entails that only one token was claimed.

For improved clarity and transparency, consider renaming the `TokenClaimed` event to `TokensClaimed`.

Update: *Resolved in pull request #1636.*

N-11 Missing Named Parameters in Mapping

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of a mapping's purpose.

In the `hasClaimed` state variable of the `ClaimableAirdrop` contract, the mapping does not have any named parameters.

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #1636.*

N-12 Unnecessary `EIP712Upgradeable` Inheritance

The `EIP712Upgradeable` contract is inherited and initialized by `ERC20PermitUpgradeable`. Therefore, there is no need to inherit `EIP712Upgradeable` directly or call `__EIP712_init` to initialize it, as this setup is already handled internally by `ERC20PermitUpgradeable`.

Consider removing the redundant inheritance and initialization to simplify the contract and avoid unnecessary code duplication. This will also make the contract easier to maintain and reduce potential errors.

Update: *Resolved in pull request #1623.*

N-13 Unnecessary Check

The allowance check in the `claim` function of the `ClaimableAirdrop` contract is unnecessary, as the `ERC20Upgradeable` contract already verifies the allowance and reverts with an informative error if it is insufficient.

Consider removing the redundant allowance check to simplify the code.

Update: *Resolved in pull request #1636.*

Conclusion

The `AlignedToken` and `ClaimableAirdrop` contracts showcase a strong and reliable design, effectively facilitating token distribution through the airdrop mechanism. The inclusion of key safeguards, such as the ability to pause and resume critical functionalities, enhances their resilience. Ensuring the accurate generation and verification of the Merkle root is essential, as it serves as the foundation for secure on-chain token claims.

The Aligned team demonstrated exceptional professionalism and collaboration throughout the audit process, promptly addressing questions and sharing updates. We applaud their commitment to security and look forward to their ongoing success.