AI Layer Airdrop Analysis

June 21, 2021

A smart contract assessment of the AI Layer Airdrop contract was performed by Octane Security, with a focus on the security aspects of the application's implementation.

About Octane Security

Octane Security is an AI-enhanced blockchain security firm dedicated to safeguarding digital assets. We combine security research with cutting-edge machine learning models to maximize protocol security and minimize risk. Our team ranks in the top of audit contest leaderboards, captures bug bounties, and has served big-name clients such as Optimism, Notional Finance, Blast, and Graph Protocol in the past.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We also rely on information that is provided by the client, its affiliates and partners for vulnerability identification. We can not guarantee the absense of vulnerabilities, issues, flaws, or defects after the review. It is up to the client to decide whether to implement recommendations and we take no liability for invalid recommendations. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Security Assessment Summary

Review commit - 0926b2b9615ca81a30f9fb806165a212222d56bd

Scope

The following smart contracts were in scope of the audit:

- `Airdrop.sol`
- `MerkleWhitelistedUpgradeable.sol`

Findings

Titles

IDN	Title	Severity
L-1	Emitting the proof bytes32 array unnecessarily increases the user's transaction cost when claiming their airdrop	Low
L-2	The function isWhitelistedUser can be removed	Low
L-3	Merkle root can be changed after initialization	Low
R-1	Usage of Ownable instead of Ownable2Step	Recommendation
R-2	Airdrop deployment will fail if using the latest version of OZ ownable contract	Recommendation
R-3	Floating Pragma	Recommendation
R-4	Suboptimal Airdrop Mock used for testing	Recommendation
R-5	Updating documentation	Recommendation

Detailed Findings

L1 - Emitting the proof bytes32 array unnecessarily increases the user's transaction cost when claiming their airdrop.

Context: `Airdrop.sol`

Description: When a user calls the `claim` function, they need to pass in a long array of bytes32 as a merkle proof. The merkle proof data is used to verify that the user is eligible to claim their reward...

```
    function claim(uint256 amount_, bytes32[] calldata merkleProof_) external {
        uint256 tokensClaimed_ = tokensClaimed[_msgSender()];

        require(tokensClaimed_ < amount_, "Airdrop: already claimed");
        require(isEligible(_msgSender(), amount_, merkleProof_), "Airdrop: not eligible

        tokensClaimed[_msgSender()] = amount_;

        IToken(token).mint(_msgSender(), amount_ - tokensClaimed_);

  @      emit Claimed(_msgSender(), amount_, merkleProof_);
    }
```

However, at the end of this function, the merkleProof_ data is emitted as well.

The bytes32[] calldata merkleProof_ may be large and the user has to pay an event cost for each additional indexed parameter in the array.

Then emitting the proof bytes32 array unnecessarily increases the user's transaction cost when claiming their airdrop.

If the protocol intends to deploy Airdrop.sol solely on an L2 where the gas fee is low then this can be acknowledged but according to the docs, if the token is distributed on mainnet as well, this issue should be fixed because the transaction cost on mainnet can be larger.

Recommendation

Write a foundry test to test the gas costs on various chains. If the gas costs are too high, consider removing the merkleProof_ from the event...

```
 emit Claimed(_msgSender(), amount_);
```

L2 - The function isWhitelistedUser can be removed.

Context: `MerkleWhitelistedUpgradeable.sol`

Description: The contract MerkleWhitelistedUpgradeable inherits from a solidity-lib dependency contract MerkleWhitelisted. The contract MerkleWhiteListed has a function isWhitelistedUser.

```
  function isWhitelistedUser(
        address user_,
        bytes32[] memory merkleProof_
    ) public view returns (bool) {
        return isWhitelisted(keccak256(abi.encodePacked(user_)), merkleProof_);
    }
```

The user may assume that they can use this function to check if a user is whitelisted and eligible for airdrop.

However, this is not the case because the leaf node is generated from user address + amount when a user is claiming their airdrop...

```
function isEligible(address user_, uint256 amount_, bytes32[] calldata merkleProof_) pu
    bytes32 leaf_ = _computeLeaf(user_, amount_);

    return isWhitelisted(leaf_, merkleProof_);
}
```

Given that the leaf node is not generated solely from the user's address as this function expects, the function isWhitelistedUser is misleading.

Recommendation: Consider overriding the isWhitelistedUser function in the MerkleWhitelistedUpgradeable contract to prevent users from mistakenly using it.

```
  function isWhitelistedUser(
        address user_,
        bytes32[] memory merkleProof_
    ) public override view returns (bool) {
        revert("function not used");
    }
```

L3 Merkle root can be changed after initialization

Context: `Airdrop.sol`

Description: The contract owner can change the Merkle root for the airdrop at any time he wants. This is a centralization risk and is a more uncommon implementation of an airdrop, because normally the merkle tree is meant to be constructed in advance and stay immutable once set - this guarantees the integrity of the Aidrop itself. The best example of this is Uniswaps `MerkleDistributor` contract, which is the `de-facto` standard for this type of airdrops:

```
contract MerkleDistributor is IMerkleDistributor {

    bytes32 public immutable override merkleRoot;

    constructor(address token_, bytes32 merkleRoot_) public {
            ....
            merkleRoot = merkleRoot_;
    }
    ....
}
```

In the MerkleDistributor above by Uniswap, the merkleRoot is set once in the constructor and forever remains immutable.

However, in Airdrop.sol, the merkleRoot can be changed at any time by the owner:

```
function setMerkleRoot(bytes32 merkleRoot_) public onlyOwner {
        _setMerkleRoot(merkleRoot_);

        emit MerkleRootSet(merkleRoot_);
    }
```

Updating the merkle tree once the Airdrop has started could complicate the airdrop process, for example all previously generated proofs would be invalid and new ones have to be regenarated (claim transactions sitting in the mempool before the `setMerkleRoot` call would all revert as well). Additionally in case the owner address gets compromised, the exploiter could break the tree or even substitute it with his own malicious version, allowing him to mint an unlimited amount of `THINK` tokens.

Recommendation: Consider removing the permissioned `setMerkleRoot` function and only call `_setMerkleRoot(merkleRoot_)` once in the initializer function. This would also prevent the hypotetical governance risk of the owner getting compromised.

R1 - Usage of Ownable instead of Ownable2Step

Context: `Airdrop.sol`

Description: The commonly used Openzeppelin Ownable implementation has a shortcoming that it allows the owner to transfer ownership to a non-existent or mistyped address. Ownable2Step is safer than Ownable because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

Recommendation: Consider using Ownable2StepUpgradeable instead of OwnableUpgradeable. This makes ownership transfers much safer and prevents accidental transfer to invalid addresses

Note: If you decide to go with Ownable2StepUpgradeable make sure to call `__Ownable_init(newOwner)` in the initializer of `Airdrop.sol`, because since v 5.0 `Ownable2StepUpgradeable` does not call it in `__Ownable2Step_init()` and would leave the contract without owner

R2 - Airdrop deployment will fail if using the latest version of OZ ownable contract

Context: `Airdrop.sol`

Description: THe Airdrop contract inherits `OwnableUpgradeable` from OZ and initializes it like so:

```
  function Airdrop_init(address token_, bytes32 merkleRoot_) external initializer {
        ....
        __Ownable_init();
        ....
    }
```

This relies on the old version of `OwnableUpgradeable` contracts. If the latest version is used (e.g v5.0) the initialization will fail, because OZ changed `__Ownable_init()` to take an argument for the owner instead of using `msg.sender`:

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/OwnableUpgradeable.sol#L51-L52

```
 function __Ownable_init(address initialOwner) internal onlyInitializing {
        __Ownable_init_unchained(initialOwner);
    }
```

This would fail early during deployment and not cause any significant issues, but still it will distort contract deployment process.

Recommendation: Consider upgrading to the latest version of OZ contracts and update `__Ownable_init() -> __Ownable_init(newOwner)`

R3 - Floating Pragma

Context: `Airdrop.sol` , `MerkleWhitelistedUpgradeable.sol`

Description: When deploying smart contracts, it is important to ensure that they are deployed using the same compiler version and flags that they have been tested with. This helps to ensure that the contracts function as intended and do not introduce bugs or vulnerabilities into the system. One way to help ensure this is to "lock" the pragma version used in the contract, which prevents it from being compiled using an outdated or incompatible compiler version.

Recommendation: Remove the `^` from the pragma.

R4 - Suboptimal Airdrop Mock used for testing

Context: `AirdropV2.sol`

Description: `AirdropV2` is used as a mock contract to test that the Aidrop proxy successfully upgrades from one version to another. However the `AirdropV2` is too basic and does not implement the actual logic of the Airdrop contract.

Recommendation: In an actual upgrade V2 would inherit from V1 and add the additional functionality. Consider updating the Mock so that it simulates the actual conditions, which would make tests more plausable.

```
contract AirdropV2 is Airdrop {
    .....
}
```

R5 - Updating documentation

Context: `README.md`

Description: While this file was out of scope, it is worth noting that you should add documentation about the `Airdrop.sol` contract to your readme as you've done for Token, Splitter, LinearDistributionIntervalDecrease etc.

Recommendation: Add the following bullet to the readme: *Airdrop* - The airdrop contract that allows users to mint their THINK tokens.