Defi App Security Review

Pashov Audit Group

Conducted by: Hals, Shaka, 0xbepresent

April 9th 2025 - April 10th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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 can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **defi-app/defi-app-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Defi App

DefiApp is a platform for managing DeFi assets across chains, offering features like native account abstraction and gasless transactions. The scope was focused on VAirdrop contract which extends Airdrop.sol with logic to create a vesting after airdrop claim and optionally stake the claimed tokens.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. 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.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 90c543f76a102d1adba66d88cc9ae1e8cd964244

fixes review commit hash - 94ddbaca8026e9d8d01b907155f41d9082b3cdc7

Scope

The following smart contracts were in scope of the audit:

- `VAirdrop`
- `Airdrop`

7. Executive Summary

Over the course of the security review, Hals, Shaka, 0xbepresent engaged with Defi App to review Defi App. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	Defi App
Repository	https://github.com/defi-app/defi-app-contracts
Date	April 9th 2025 - April 10th 2025
Protocol Type	Airdrop and Vesting contract

Findings Count

Severity	**Amount**
High	1
Medium	2
Low	3
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Accounts with two claimable leaves can claim just one	High	Resolved
[M-01]	canClaim() and claim() compute leaf value differently	Medium	Resolved
[M-02]	Inconsistent leaf calculation causes canClaim mismatch	Medium	Resolved
[L-01]	canClaim() lacks maximum claimable amount check	Low	Resolved
[L-02]	Missing endTimestamp validation in constructor	Low	Resolved
[L-03]	Owner bypasses withdrawal delay by setting endTimestamp in the past	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Accounts with two claimable leaves can claim just one

Severity

Impact: Medium

Likelihood: High

Description

According to the PR comments it is expected that some accounts will be able to use two sets of leaf params, which will allow them to claim a portion of the tokens instantly and another portion of the tokens with a vested period.

However, with the current implementation, the second claim will fail, as claims are tracked by account, and not by leaf, so the same account cannot claim more than once.

The tokens to be claimed might not necessarily be lost, as the owner of the contract can recover them after the airdrop is finished and create a new airdrop contract with the new leaf params (thus the medium impact severity), but this is far from ideal, as it forces the recipient to wait for the first airdrop to be finished and the owner to provide a new contract.

Recommendations

Use the leaf to track claims instead of the account address.

8.2. Medium Findings

[M-01] `canClaim()` and `claim()` compute leaf value differently

Severity

Impact: Low

Likelihood: High

Description

The `canClaim()` function in the `Airdrop` contract computes the leaf value differently from how it is computed in the `claim()` function.

```
function claim(bytes memory leafElements, bytes32[] calldata merkleProof)
(...)
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender, amount));

(...)

    function canClaim(
      addressuser,
      uint256amount,
      bytes32[]calldatamerkleProof
    ) external view returns (bool
(...)
            bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode
              (user, amount))));
```

As a result, `canClaim()` will always return `false` even for valid claims. This might cause confusion for users who believe they are eligible to claim tokens and might ultimately not initiate the claim process, losing their tokens.

Recommendations

```
function canClaim(
      addressuser,
      uint256amount,
      bytes32[]calldatamerkleProof
    ) external view returns (bool
        if (block.timestamp <= endTimestamp && !hasClaimed[user]) {
            // Compute the leaf and verify the merkle proof
-           bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode
- (user, amount))));
+           bytes32 leaf = keccak256(abi.encodePacked(user, amount));
```

[M-02] Inconsistent leaf calculation causes `canClaim` mismatch

Severity

Impact: Medium

Likelihood: Medium

Description

In the provided contracts, the `VAirdrop` contract inherits from `Airdrop` without overriding the `canClaim` function. The issue stems from the difference in how the Merkle leaf is calculated in each context. The `Airdrop` contract computes the leaf as follows:

```
File: Airdrop.sol
100:     function canClaim(
  addressuser,
  uint256amount,
  bytes32[]calldatamerkleProof
) external view returns (bool
101:         if (block.timestamp <= endTimestamp && !hasClaimed[user]) {
102:             // Compute the leaf and verify the merkle proof
103:             bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode
  (user, amount))));
104:             return MerkleProof.verify(merkleProof, merkleRoot, leaf);
105:         } else {
106:             return false;
107:         }
108:     }
```

Meanwhile, the `VAirdrop` contract's claim function computes the leaf using an extended structure that includes additional fields (`smartAccount` and `vesting`):

```
File: VAirdrop.sol
64:         // Compute the leaf and verify the merkle proof
65:         bytes32 leaf = keccak256(abi.encodePacked
  (msg.sender, amount, smartAccount, vesting));
66:         if (!MerkleProof.verify
  (merkleProof, merkleRoot, leaf)) revert InvalidProof();
```

Because the `canClaim` function in `Airdrop` is not overwritten in `VAirdrop`, it uses the leaf format with only `user` and `amount`. This results in a mismatch:

- When users check `canClaim`, the computed leaf does not match the leaf used during the actual claim process.
- As a consequence, `canClaim` always returns false even when a valid proof exists, leading to incorrect eligibility status reporting.

The next test helps to reproduce the issue when the `canClaim` function always return `false`, but the `claim` function works as expected.

```
function test_canClaimReturnsAlwaysFalse() public {
        vm.prank(Admin.addr);
        airdrop.setMerkleRoot(vairdropRoot);
        bytes memory leafElements = abi.encode
          (USER1_AMOUNT_TO_RECEIVE, User1.addr, false);

        assertEq(homeToken.balanceOf(User1.addr), 0);
        vm.prank(User1.addr);
        //
        // canClaim returns `false`
        bool canClaim = airdrop.canClaim(address
          (User1.addr), USER1_AMOUNT_TO_RECEIVE, user1vairdropProof);
        assertFalse(canClaim);
        vm.stopPrank();
        //
        // However, the claim is successful
        vm.prank(User1.addr);
        airdrop.claim(leafElements, user1vairdropProof);
        assertEq(homeToken.balanceOf(User1.addr), USER1_AMOUNT_TO_RECEIVE);
    }
```

Recommendations

Update the `canClaim` function in the `VAirdrop` contract so that it computes the leaf using the same structure as in the `VAirdrop::claim` function. By overriding the `canClaim` function in `VAirdrop` to match the leaf calculation used in the `VAirdrop::claim` process, this discrepancy is resolved and both functions will operate consistently.

8.3. Low Findings

[L-01] `canClaim()` lacks maximum claimable amount check

The `canClaim()` function in `Airdrop.sol` does not check if the amount to be claimed exceeds the maximum claimable amount per user. This check is performed in the `claim()` function, so it is possible that `canClaim()` returns true even if the user is not eligible to claim the requested amount.

It is recommended to introduce the following changes to the codebase:

```
function canClaim(
      addressuser,
      uint256amount,
      bytes32[]calldatamerkleProof
    ) external view returns (bool
-       if (block.timestamp <= endTimestamp && !hasClaimed[user]) {
+       if
+ (block.timestamp <= endTimestamp && !hasClaimed[user] && amount <= MAXIMUM_AMOUNT_TO
```

[L-02] Missing `endTimestamp` validation in constructor

The Airdrop contract's constructor fails to validate whether the provided `_endTimestamp` is within the acceptable limits (i.e., not in the past and within 30 days of the current block timestamp). In contrast, the `updateEndTimestamp` function validates the new timestamp by ensuring it is not more than `block.timestamp + 30 days`.

```
File: Airdrop.sol
49:       */
50:      constructor(
  uint256_endTimestamp,
  uint256_maximumAmountToClaim,
  address_distributionToken
) Ownable(msg.sender
51:@>        endTimestamp = _endTimestamp;
...
144:      function updateEndTimestamp
  (uint256 newEndTimestamp) external onlyOwner {
145:@>        if
  (block.timestamp + 30 days < newEndTimestamp) revert NewTimeStampTooFar();
146:          endTimestamp = newEndTimestamp;
147:
148:          emit NewEndTimestamp(newEndTimestamp);
149:      }
```

This inconsistency results in two scenarios:

- The contract can be deployed with an `endTimestamp` that is already expired. As a result, functions such as `claim` will always revert due to the timestamp condition.

```
function test_constructorEndTimestampPast() public {
        uint256 pastTimestamp = 0;

        vm.startPrank(Admin.addr);
        VAirdrop airdropPast = VAirdropDeployer.deploy(
            fs,
            "VAirdrop",
            TESTING_ONLY,
            VAirdropInitParams({
                vestingManager: address(vesting),
                stakingContract: address(staking),
                endTimestamp: pastTimestamp,
                maximumAmountToClaim: maxClaimableAmount,
                distributionToken: address(homeToken)
            })
        );
        airdropPast.setMerkleRoot(vairdropRoot);
        vm.stopPrank();

        bytes memory leafElements = abi.encode
          (USER1_AMOUNT_TO_RECEIVE, User1.addr, false);
        assertEq(homeToken.balanceOf(User2.addr), 0);
        //
        // claim revert since the end timestamp is in the past
        vm.prank(User1.addr);
        vm.expectRevert(Airdrop.ClaimTimeExceeded.selector);
        airdropPast.claim(leafElements, user1vairdropProof);
    }
```

- The constructor does not validate that the `endTimestamp` is within 30 days. Meanwhile, the `updateEndTimestamp` function explicitly reverts if the new timestamp exceeds `block.timestamp + 30 days`.

```
function test_constructorEndTimestampFuture() public {
        uint256 futureTimestamp = block.timestamp + 60 days;

        vm.startPrank(Admin.addr);
        VAirdrop airdropFuture = VAirdropDeployer.deploy(
            fs,
            "VAirdrop",
            TESTING_ONLY,
            VAirdropInitParams({
                vestingManager: address(vesting),
                stakingContract: address(staking),
                endTimestamp: futureTimestamp,
                maximumAmountToClaim: maxClaimableAmount,
                distributionToken: address(homeToken)
            })
        );
        //
        // end timestamp is in the future
        assertEq(airdropFuture.endTimestamp(), futureTimestamp);
        //
        // Revert when updating the end timestamp to a future date using the
        // function
        vm.expectRevert(Airdrop.NewTimeStampTooFar.selector);
        airdropFuture.updateEndTimestamp(futureTimestamp);
    }
```

Modify the constructor to require that `_endTimestamp` is both in the future and not more than 30 days from the current block time.

[L-03] Owner bypasses withdrawal delay by setting `endTimestamp` in the past

The `updateEndTimestamp()` function allows the `Airdrop` contract owner to set the `endTimestamp` to any value within 30 days into the future relative to the current `block.timestamp`. However, it does not enforce that the new `endTimestamp` must be in the future. This means the owner can set the `endTimestamp` to a time in the past, which would immediately disable the ability for users to claim airdrops.

This behavior could be abused by a malicious owner to:

- Prematurely end the claim window, preventing legitimate users from claiming their airdropped tokens.
- Circumvent the `withdrawTokenRewards()` safeguard that requires waiting 1 day after the claim period ends before tokens can be withdrawn, effectively shortening the reward tokens withdrawal delay.

```
function updateEndTimestamp(uint256 newEndTimestamp) external onlyOwner {
        if
          (block.timestamp + 30 days < newEndTimestamp) revert NewTimeStampTooFar();
        endTimestamp = newEndTimestamp;

        emit NewEndTimestamp(newEndTimestamp);
    }
```

```
function withdrawTokenRewards() external onlyOwner {
        if (block.timestamp <
          (endTimestamp + 1 days)) revert TooEarlyToWithdraw();
        uint256 balanceToWithdraw = distributionToken.balanceOf(address(this));
        distributionToken.safeTransfer(msg.sender, balanceToWithdraw);

        emit TokensWithdrawn(balanceToWithdraw);
    }
```

Recommendation: Update the `updateEndTimestamp()` function to include a check that ensures the new `endTimestamp` is strictly greater than the current `block.timestamp`.