DropClaim Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

carlitox477

Assisting Auditors

Alex Roan

June 13, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The Dropclaim contract is a versatile solution for claiming and batch claiming airdrops. It provides options for contract owners to execute airdrops in either a fee-based or allowlist-based mode. In the fee-based mode, claimers can claim airdrops by paying a minimum fee, whereas in the allowlist-based mode, only whitelisted claimers can make claims without any fee.

The contract offers the functionality for owners to transfer ownership to a new owner who then needs to claim their ownership. This ensures a secure transition of ownership within the contract.

4.1 Key actors & actions

There are two key actors involved in this contract: claimers and contract owners. Claimers have the ability to make single or batch claims on pre-approved claim contracts. In the fee-based mode, claimers are required to pay a minimum fee in ETH to specific fee recipients before their claims are considered valid. On the other hand, in the allowlist mode, claimers who are part of the allowlist need to submit valid merkle proofs, which are then verified by the contract. No fee payment is necessary in this mode.

Contract owners hold special privileges, including the ability to set claim parameters and the merkle root in the case of a user allowlist. They can also designate fee recipients for fee-based claims. Owners have the flexibility to add, update, and remove claims at any time. For fee-based airdrops, owners can modify the minimum fees required from claimers. Similarly, for allowlist-based airdrops, owners can update the merkle root that serves as proof of claimer eligibility. In both cases, claims have an expiry date, and claimers must make their claims before the expiration. In fee-based airdrops, owners can assign one or multiple fee recipients who will receive the fees paid by claimers during the airdrop.

The Dropclaim contract provides a robust and customizable solution for executing airdrops with different claim modes, offering flexibility for both claimers and contract owners.

4.2 Security Remarks

1. High Trust Setup: The current setup heavily relies on trust in contract owners. They possess unrestricted powers to add, delete, and update claims, as well as modify fee recipients and merkle roots that verify claimer eligibility. It is important to address the high trust levels associated with the contract owners. We recommend implementing improvements that reduce the trust required to a reasonable level.
2. Improved Documentation: While the `DropClaim` contract abstracts the core claim logic, acting as an airdrop primitive, it delegates security considerations to the claim contracts. The `DropClaim` contract primarily ensures correct payments and allows only whitelisted accounts to make claims, without addressing issues like duplicate claims. While this reduces the attack surface of the `DropClaim` contract, it increases the security responsibility for any claim contracts utilizing this setup. It is crucial to enhance the existing documentation to clearly communicate the security expectations for claim contracts using the `DropClaim` setup.

5 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the DropClaim smart contracts provided by Bankless DAO. In this period, a total of 5 issues were found.

Summary

Project Name	DropClaim
Repository	drop-claim
Commit	e411390a4626...
Audit Timeline	May 22nd - May 29th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	2
Informational	1
Gas Optimizations	2
Total Issues	5

Summary of Findings

[L-1] An expired claim can be revived by a contract owner well past the expiry date	Acknowledged
[L-2] Changing minimum fee and expiry midway through an active airdrop can be unfair to existing/future claimers	Acknowledged

[I-1] `allowlistClaim` allows users to access `claimContract` unlimited times creating a potential for duplicate calls	Acknowledged
[G-1] Use storage pointer rather than copy in memory	Resolved
[G-2] Bool comparison to constant values should be avoided	Resolved

6 Findings

6.1 Low Risk

6.1.1 An expired claim can be revived by a contract owner well past the expiry date

Description: Contract owners can use the `setClaims` function to configure claim parameters for each claim contract, including settings such as `expiry` and `minFee`. For a claim to be considered valid, claimers must pay a fee that exceeds the specified `minFee` and claim the expiry date.

However, it is important to note that the `setClaims` function allows contract owners to add or update a claim with an expiry date that has already passed. While this may initially seem like a harmless side-effect, as claimers cannot claim on such contracts, it has a more significant implication. Contract owners can effectively revive airdrops that have already expired, enabling them to carry out airdrops that should no longer be active.

Affected lines of code in `DropClaims::setClaims`

Impact: Contract owners can modify a claim even after it has expired. This allows retroactively adding claimers by extending the validity period of a claim that should have already ended.

Recommended Mitigation: While the `setClaims` function is limited to contract owners and certain trust assumptions are made by protocol developers, we recommend implementing additional controls to mitigate the unrestricted powers granted to contract owners.

Consider validating the claim time before adding/updating an existing claim.

```
// DropClaim::setClaims
    for (uint256 i; i < arrLength;) {
+       if(expiries[i] <= block.timsetstamp) revert ERROR_EXPIRE_TIME_SHOULD_BE_GREATER_THAN_NOW;
        claims[claimContractHashes[i]] = ClaimData(uint64(expiries[i]), uint128(minFees[i]));

        unchecked {
            ++i;
        }
    }
```

Bankless: Acknowledged. Team will not make any changes related to this finding as this is intended functionality.

Cyfrin: Acknowledged.

6.1.2 Changing minimum fee and expiry midway through an active airdrop can be unfair to existing/future claimers

Description: `DropClaim::setClaims` allows users to overwrite claim parameters, `expiry`, and `minimumFee` of an existing claim contract hash.

```
function setClaims(bytes32[] calldata claimContractHashes, uint256[] calldata expiries, uint256[]
↪  calldata minFees)
      external
      onlyOwner
  {
      if (claimContractHashes.length != expiries.length) revert MismatchedArrayLengths();
      if (claimContractHashes.length != minFees.length) revert MismatchedArrayLengths();

      uint256 arrLength = claimContractHashes.length;
      for (uint256 i; i < arrLength;) {
          claims[claimContractHashes[i]] = ClaimData(uint64(expiries[i]), uint128(minFees[i]));
          // @audit -> claim parameters for an existing claim contract hash can be overwritten for an
          ↪  active airdrop
          unchecked {
              ++i;
          }
      }
  }
```

When users commence claiming airdrops, it is important to consider the potential consequences of following actions that the protocol may initiate:

1. Adjusting the validity period for future claimers by extending or reducing it.
2. Modifying the fees for future claimers by increasing or decreasing them.

In both cases, it is crucial to recognize that such actions have the potential to create unfairness, whether it is for existing claimers or future claimers. Even the protocol owners should not be able to manipulate airdrop parameters once the airdrop has been activated.

In the case of using the `allowList` mode for claims, a similar concern arises regarding the `setMerkleRoot` function. Owners can update the Merkle root even after the airdrop has been activated and some users have made their claims.

Impact: A decrease in `minFees` benefits future claimers, while an increase in `minFees` benefits existing claimers. Similarly, shortening validity benefits past claimers while extending validity benefits future claimers. Allowing owners to update the Merkle root once the airdrop is activated has the potential to render previously eligible claimers ineligible and vice versa.

Recommended Mitigation: Consider tracking an additional parameter, `numClaimers`, in the `ClaimData` struct. Note that even with this addition, `ClaimData` still fits in a single slot.

```
  struct ClaimData {
        uint64 expiry; // Timestamp beyond which claims are disallowed
        uint128 minFee; // Minimum ETH fee amount
+        uint64 numClaimers;// Number of users who already claimed //@audit -> add this variable to
↪  ClaimData struct
    }
```

Increment `numClaimers` every time a new claim is successful. Allow changes in `setClaims` and `setMerkleRoot` only if no existing claimers exist.

`DropClaim::setClaims`

```
    function setClaims(bytes32[] calldata claimContractHashes, uint256[] calldata expiries, uint256[]
    ↪  calldata minFees)
        external
        onlyOwner
    {
        if (claimContractHashes.length != expiries.length) revert MismatchedArrayLengths();
        if (claimContractHashes.length != minFees.length) revert MismatchedArrayLengths();

        uint256 arrLength = claimContractHashes.length;
        for (uint256 i; i < arrLength;) {
+         require(claims[claimContractHashes[i]].numClaimers ==0, "Airdrop already activated");
            claims[claimContractHashes[i]] = ClaimData(uint64(expiries[i]), uint128(minFees[i]));
            unchecked {
                ++i;
            }
        }
    }
```

Bankless: Acknowledged. Team will not change as is worth the trade-off for us.

Cyfrin: Acknowledged.

6.2 Informational

6.2.1 `allowlistClaim` allows users to access `claimContract` unlimited times creating a potential for duplicate calls

Description: The `allowlistClaim` function enables claimers to repeatedly access the functions of the `claimContract`. However, it doesn't track whether a claimer has already submitted proof for verification. This can be a problem for contracts that involve airdrops, as it's important to limit the number of successful interactions with the `claimContract`.

Currently, checking for duplicate interactions lies solely with the `claimContract` because `dropClaim` contract only verifies Merkle proofs.

In contrast, the claim function in the MerkleDistributor contract deployed by Uniswap and 1inch handles both verification of proofs submitted by claimers and ensuring that each claimer can only have one successful call.

Here's a snippet of the Uniswap `Merkledistributor` contract at 0x090D4613473dEE047c3f2706764f49E0821D256e:

```
function claim(uint256 index, address account, uint256 amount, bytes32[] calldata merkleProof)
↪  external override {
      require(!isClaimed(index), 'MerkleDistributor: Drop already claimed.'); //@restricting each
      ↪  user to 1 successful claim
     //@audit -> this verifies if already claimed or not
      // Verify the merkle proof.
      bytes32 node = keccak256(abi.encodePacked(index, account, amount));
      require(MerkleProof.verify(merkleProof, merkleRoot, node), 'MerkleDistributor: Invalid proof.');

      // Mark it claimed and send the token.
      _setClaimed(index);
      require(IERC20(token).transfer(account, amount), 'MerkleDistributor: Transfer failed.');

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

Impact: In situations such as claiming airdrops, minting NFTs or claiming vested shares, it's crucial to limit the number of calls made by each claimer. Allowing duplicate access to key functions like `claim` without explicit handling of duplications can result in losses for the protocol. Relying solely on `claimContracts` to handle these duplications can potentially increase the attack surface.

Recommended Mitigation: We propose considering an enhancement to the `DropClaim` logic that would take into account the number of claims per `claimContract` for each user. To achieve this, we suggest introducing a mapping system that associates user addresses with the number of successful claims they have made.

In line with the existing configuration options, such as `minFee` and `expiry` parameters for each `claimContract`, we recommend the addition of a new parameter called `maxCallsPerContract`. This parameter would allow owners to limit the number of claims a user can make for a specific `claimContract`.

This added layer of security provided by the `DropClaim` contract will involve verifying if the total claims made by a user for a given `claimContract` are below the set `maxCallsPerContract` value for that particular contract. Although more gas expensive, this enhancement reduces attack surface corresponding to duplicate claims.

Bankless: Acknowledged. Team will not make any changes related to this finding as this is intended functionality.

Cyfrin: Acknowledged.

6.3 Gas Optimization

6.3.1 Use storage pointer rather than copy in memory

To avoid copying every element of the struct when only one element is required, it is more efficient to use a storage pointer rather than copy the element in memory.

```
// DropClaim::claim
-   ClaimData memory claimData = claims[getClaimContractHash(claimContract, salt)];
+   ClaimData storage claimData = claims[getClaimContractHash(claimContract, salt)];
```

Line 100

```
// DropClaim::batchClaim
-   ClaimData memory claimData = claims[getClaimContractHash(claimContracts[i], salts[i])];
+   ClaimData storage claimData = claims[getClaimContractHash(claimContracts[i], salts[i])];
```

Line 131

```
// DropClaim::_claim
-   ClaimData memory claimData = claims[getClaimContractHash(claimContract, salt)];
+   ClaimData storage claimData = claims[getClaimContractHash(claimContract, salt)];
```

Line 200

Bankless: Acknowledged & fixed in commit 020bb4c49a281af32898f951b784d7748dac049f.

Cyfrin: Verified.

6.3.2 Bool comparison to constant values should be avoided

Comparing to a constant (true or false) is a bit more expensive than directly checking the returned boolean value.

```
// DropClaim::allowlistClaim
-   if (MerkleProof.verify(merkleProof, merkleRoot, bytes32(uint256(uint160(msg.sender)))) == false) {
+   if (!MerkleProof.verify(merkleProof, merkleRoot, bytes32(uint256(uint160(msg.sender))))) {
```

Line 158

```
// DropClaim::allowlistBatchClaim
-   if (MerkleProof.verify(merkleProof, merkleRoot, bytes32(uint256(uint160(msg.sender)))) == false) {
+   if (!MerkleProof.verify(merkleProof, merkleRoot, bytes32(uint256(uint160(msg.sender))))) {
```

Line 181

Bankless: Acknowledged & fixed in commit 0d3ccd6eb7ad266be54598a52d321cb9bb17e7af.

Cyfrin: Verified

7 Appendix

7.1 Static Analysis

Cleaned output from the 4naly3er tool.

7.1.1 Gas Optimizations

	Issue	Instances
GAS-1	Using bools for storage incurs overhead	1
GAS-2	Functions guaranteed to revert when called by normal users can be marked `payable`	2

[GAS-1] Using bools for storage incurs overhead Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (1):

```
File: DropClaim.sol

49:     mapping(address => bool) internal feeRecipients;
```

[GAS-2] Functions guaranteed to revert when called by normal users can be marked `payable` If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Instances (2):

```
File: DropClaim.sol

213:     function setFeeRecipients(address[] calldata _feeRecipients, bool[] calldata enabled) external
↪  onlyOwner {

225:     function setMerkleRoot(bytes32 _merkleRoot) external onlyOwner {
```