AirDropper Audit Report

Prepared by Putfor

February 22, 2026

Table of Contents

Protocol Summary

`Airdropper` is a Merkle-proof-based distribution protocol intended to airdrop `100 USDC` on zkSync Era to four eligible addresses (`25 USDC` each). The core flow allows a user to claim tokens by submitting `(account, amount, proof)` to `MerkleAirdrop.claim`, while the contract owner can withdraw accumulated claim fees via `claimFees`.

The deployment path is handled by `Deploy.s.sol`, which initializes the Merkle root and funds the airdrop contract with the target token.

Disclaimer

Putfor makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security research by Putfor is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings in this document correspond to the following commit hash:

```
ff4fcafa1b68776adf930b507a1b3dd192de38d6
```

Scope

```
./src/
└── MerkleAirdrop.sol
./script/
└── Deploy.s.sol
```

Compatibilities

Blockchains:
- zkSync Era

Tokens:
- Native ETH (claim fee payment)
- USDC (airdrop asset)

Standards:
- ERC20 interactions for token transfers

Roles

Owner: Authorized to withdraw collected claim fees from `MerkleAirdrop`.

Claimant: Eligible recipient included in the Merkle tree who claims USDC with a valid proof and fee payment.

Executive Summary

This assessment combined manual security review with targeted automated and dynamic validation to maximize both depth and reproducibility.

The review process included:

- Manual line-by-line analysis of claim logic, access control boundaries, and deployment assumptions.
- Foundry-based unit testing for exploit reproduction and behavioral validation.
- Static analysis with Slither to surface structural risks and code-quality issues.
- Symbolic-analysis-assisted checks with Mythril to expand path coverage and detect common vulnerability patterns.

The audit identified critical correctness issues in the token distribution flow and operational risks in deployment configuration, as well as low-severity maintainability findings.

Issues found

Severity	Number of issues found
High	2
Low	2
Total	4

Findings

High

[H-1] Missing Claim-State Tracking Allows Unlimited Reuse of Merkle Proofs

Description

The `claim` function in `MerkleAirdrop` verifies that a given `(account, amount)` pair is part of the Merkle tree but does not record whether the claim has already been executed.

```
function claim(address account, uint256 amount, bytes32[] calldata merkleProof)
external payable {
    if (msg.value != FEE) {
        revert MerkleAirdrop__InvalidFeeAmount();
    }

    bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(account,
amount))));
    if (!MerkleProof.verify(merkleProof, i_merkleRoot, leaf)) {
        revert MerkleAirdrop__InvalidProof();
    }

    emit Claimed(account, amount);
    i_airdropToken.safeTransfer(account, amount);
}
```

The Merkle root is immutable and no claim-state invalidation mechanism is implemented (e.g., no mapping of claimed leaves or addresses).

As a result, a legitimate claimant can repeatedly call `claim` using the same `(account, amount, proof)` and receive tokens each time. Since the contract does not track previously claimed allocations, the same proof remains valid indefinitely.

This enables unbounded re-claims until the airdrop token balance is exhausted.

Risk

Likelihood: High

Exploitation requires no special privileges or complex setup. Any eligible user who successfully submits a valid Merkle proof can simply repeat the exact same transaction parameters and receive tokens again. The absence of claim-state invalidation makes the attack deterministic and trivial to execute.

Impact: High

A single eligible claimant can withdraw tokens far exceeding their intended allocation, potentially draining the entire airdrop pool. This directly breaks the fundamental correctness of the distribution mechanism and results in material financial loss, preventing legitimate users from receiving their allocations.

Proof of Concept

The following Foundry test demonstrates that a legitimate claimant can reuse the same Merkle proof multiple times and receive their allocation repeatedly.

```
function test_ClaimCanBeRepeatedWithSameProof() public {
    // Ensure claimant has sufficient ETH to repeatedly pay the claim fee
    vm.deal(collectorOne, 10 ether);

    vm.startPrank(collectorOne);

    // First legitimate claim
    airdrop.claim{value: airdrop.getFee()}(
        collectorOne,
        amountToCollect,
        proof
    );

    // Reuse the exact same proof multiple times
    airdrop.claim{value: airdrop.getFee()}(
        collectorOne,
        amountToCollect,
        proof
    );

    airdrop.claim{value: airdrop.getFee()}(
        collectorOne,
        amountToCollect,
        proof
    );

    airdrop.claim{value: airdrop.getFee()}(
        collectorOne,
        amountToCollect,
        proof
    );

    vm.stopPrank();
    // Collector receives allocation 4 times instead of once
    uint256 endingBalance = token.balanceOf(collectorOne);
    assertEq(endingBalance, amountToCollect * 4);
}
```

Run the test:

```
forge test --match-test test_ClaimCanBeRepeatedWithSameProof -vv
```

Output:

```
Ran 1 test for test/MerkleAirdropTest.t.sol:MerkleAirdropTest
[PASS] test_ClaimCanBeRepeatedWithSameProof() (gas: 130869)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.37ms

Ran 1 test suite in 17.27ms: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

This confirms that the contract allows unlimited reuse of the same Merkle proof, resulting in repeated token transfers.

Recommended Mitigation

The recommended approach is to implement explicit claim-state tracking rather than dynamically modifying the Merkle root.

A simple and robust solution is to track claimed allocations using a mapping:

```
+   mapping(bytes32 => bool) public claimed;
```

Then modify the `claim` function:

```
function claim(address account, uint256 amount, bytes32[] calldata merkleProof)
external payable {
    if (msg.value != FEE) {
        revert MerkleAirdrop__InvalidFeeAmount();
    }

    bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(account,
amount))));

+   if (claimed[leaf]) {
+       revert MerkleAirdrop__AlreadyClaimed();
+   }

    if (!MerkleProof.verify(merkleProof, i_merkleRoot, leaf)) {
        revert MerkleAirdrop__InvalidProof();
    }

+   claimed[leaf] = true;

    emit Claimed(account, amount);
    i_airdropToken.safeTransfer(account, amount);
}
```

This ensures that each valid leaf can only be claimed once, preserving the integrity of the airdrop distribution while maintaining a static Merkle root.

[H-2] Incorrect Hardcoded Token Address in Deployment Script Can Break Airdrop Funding

Description

The deployment script responsible for initializing and funding the `MerkleAirdrop` contract contains an inconsistency in the token address used for transferring funds.

```
contract Deploy is Script {
    address public s_zkSyncUSDC = 0x1D17CbCf0D6d143135be902365d2e5E2a16538d4;
    bytes32 public s_merkleRoot =
0xf69aaa25bd4dd10deb2ccd8235266f7cc815f6e9d539e9f4d47cae16e0c36a05;
    uint256 public s_amountToAirdrop = 4 * (25 * 1e6);

    function run() public {
        vm.startBroadcast();
        MerkleAirdrop airdrop = deployMerkleDropper(s_merkleRoot,
IERC20(s_zkSyncUSDC));

        // Hardcoded address does not match s_zkSyncUSDC
        IERC20(0x1d17CBcF0D6D143135aE902365D2E5e2A16538D4)
            .transfer(address(airdrop), s_amountToAirdrop);

        vm.stopBroadcast();
    }
}
```

Although the script correctly deploys the `MerkleAirdrop` contract using `s_zkSyncUSDC`, it transfers tokens from a separately hardcoded address that differs from the declared variable.

As a result, the airdrop contract may end up receiving tokens from an unintended contract while the intended token (`s_zkSyncUSDC`) is never transferred. This can leave the airdrop deployed in an unfunded or unusable state. In the worst case, tokens may be mistakenly transferred from or to an unintended contract, creating operational confusion and requiring redeployment.

Risk

Likelihood: High

Deployment scripts are executed manually or during CI/CD processes, and such inconsistencies are easy to overlook, especially when addresses visually resemble each other. Since both addresses appear similar, the mistake may not be immediately detected before broadcasting the transaction.

Impact: High

If the incorrect token address is used during funding, the deployed airdrop contract may remain unfunded or funded with the wrong asset, breaking the distribution process. While this does not directly compromise contract security, it can lead to failed launches, incorrect token transfers, or operational disruption requiring redeployment.

Proof of Concept

The issue is directly observable in the deployment script:

- `s_zkSyncUSDC` is defined as:

```
0x1D17CbCf0D6d143135be902365d2e5E2a16538d4
```

- The actual transfer uses:

```
0x1d17CBcF0D6D143135aE902365D2E5e2A16538D4
```

Since these addresses are not identical, the funding step does not reference the same token contract that was used during deployment.

This confirms that the script does not consistently use the intended token address.

Recommended Mitigation

Avoid hardcoded duplicate addresses and consistently reference the declared variable throughout the deployment script.

Replace:

```diff
-   IERC20(0x1d17CBcF0D6D143135aE902365D2E5e2A16538D4)
        .transfer(address(airdrop), s_amountToAirdrop);
```

With:

```diff
+   IERC20(s_zkSyncUSDC)
        .transfer(address(airdrop), s_amountToAirdrop);
```

Or, even better, reuse the already typed variable:

```
IERC20 token = IERC20(s_zkSyncUSDC);
token.transfer(address(airdrop), s_amountToAirdrop);
```

This approach establishes a single source of truth for the token address and eliminates the risk of silent copy-paste inconsistencies. It also improves the overall safety and maintainability of the deployment logic by ensuring that all references to the token are derived from the same variable.

Low

[L-1] Unused `MerkleRootUpdated` Event in `MerkleAirdrop`

Description

`MerkleAirdrop` declares the following event:

```
event MerkleRootUpdated(bytes32 newMerkleRoot);
```

However, this event is never emitted anywhere in the contract.

At the same time, `i_merkleRoot` is immutable and there is no function that updates the Merkle root, so this event is currently dead code.

Keeping unused events in production contracts can create confusion for integrators and auditors, because it implies behavior (root updates) that does not actually exist.

Risk

Likelihood: High

The issue is definitely present in the current codebase: the event exists but has no emit path.

Impact: Low

This does not create direct fund-loss risk or privilege escalation. The impact is limited to code clarity, maintainability, and potential monitoring/indexing confusion.

Proof of Concept

Static check confirms the event is declared once and never emitted:

```
rg -n "MerkleRootUpdated|emit\s+MerkleRootUpdated" src test
```

Output:

```
src/MerkleAirdrop.sol:20:    event MerkleRootUpdated(bytes32 newMerkleRoot);
```

No `emit MerkleRootUpdated(...)` usage is found.

Thus, the finding is confirmed: the event is unused.

Recommended Mitigation

Remove the unused event declaration to keep the contract interface minimal and accurate.

```diff
-    event MerkleRootUpdated(bytes32 newMerkleRoot);
```

If Merkle root rotation is planned in future versions, keep the event only when a real root-update flow is implemented and emits it.

[L-2] Hardcoded Deployment Parameters Are Mutable Storage Instead of Constants

Description

In `script/Deploy.s.sol`, the deployment configuration is hardcoded as state variables:

```
address public s_zkSyncUSDC = 0x1D17CbCf0D6d143135be902365d2e5E2a16538d4;
bytes32 public s_merkleRoot =
0xf69aaa25bd4dd10deb2ccd8235266f7cc815f6e9d539e9f4d47cae16e0c36a05;
uint256 public s_amountToAirdrop = 4 * (25 * 1e6);
```

These values are never updated and are only used as fixed deployment parameters.

Keeping immutable configuration as mutable storage is unnecessary and reduces clarity. It also weakens intent signaling: readers cannot immediately distinguish truly fixed constants from values that may be changed in future script flows.

Additionally, `run()` uses a second hardcoded USDC literal in `transfer(...)` instead of reusing `s_zkSyncUSDC`, which introduces avoidable duplication.

Risk

Likelihood: High

This is definitely present in the codebase and always affects maintainability/readability of the deployment script.

Impact: Low

The issue does not directly create an exploitable on-chain vulnerability in the deployed `MerkleAirdrop` contract. The impact is limited to code quality, maintainability, and configuration hygiene.

Proof of Concept

The variable declarations in `Deploy.s.sol` are storage-backed and non-constant:

```
rg -n "s_zkSyncUSDC|s_merkleRoot|s_amountToAirdrop" script/Deploy.s.sol
```

Output:

```
8:    address public s_zkSyncUSDC = 0x1D17CbCf0D6d143135be902365d2e5E2a16538d4;
9:    bytes32 public s_merkleRoot =
0xf69aaa25bd4dd10deb2ccd8235266f7cc815f6e9d539e9f4d47cae16e0c36a05;
11:    uint256 public s_amountToAirdrop = 4 * (25 * 1e6);
16:        MerkleAirdrop airdrop = deployMerkleDropper(s_merkleRoot,
IERC20(s_zkSyncUSDC));
18:
IERC20(0x1d17CBcF0D6D143135aE902365D2E5e2A16538D4).transfer(address(airdrop),
s_amountToAirdrop);
```

Thus, fixed deployment values are currently stored as mutable state variables, and the token address is duplicated in two different literals/usages.

Recommended Mitigation

Declare fixed deployment parameters as `constant` and reuse them consistently.

```
 contract Deploy is Script {
-    address public s_zkSyncUSDC = 0x1D17CbCf0D6d143135be902365d2e5E2a16538d4;
-    bytes32 public s_merkleRoot =
0xf69aaa25bd4dd10deb2ccd8235266f7cc815f6e9d539e9f4d47cae16e0c36a05;
+    address public constant ZKSYNC_USDC =
0x1D17CbCf0D6d143135be902365d2e5E2a16538d4;
+    bytes32 public constant MERKLE_ROOT =
0xf69aaa25bd4dd10deb2ccd8235266f7cc815f6e9d539e9f4d47cae16e0c36a05;
     // 4 users, 25 USDC each
-    uint256 public s_amountToAirdrop = 4 * (25 * 1e6);
+    uint256 public constant AMOUNT_TO_AIRDROP = 4 * (25 * 1e6);

     function run() public {
         vm.startBroadcast();
-        MerkleAirdrop airdrop = deployMerkleDropper(s_merkleRoot,
IERC20(s_zkSyncUSDC));
+        MerkleAirdrop airdrop = deployMerkleDropper(MERKLE_ROOT,
IERC20(ZKSYNC_USDC));
         // Send USDC -> Merkle Air Dropper
-
IERC20(0x1d17CBcF0D6D143135aE902365D2E5e2A16538D4).transfer(address(airdrop),
s_amountToAirdrop);
+        IERC20(ZKSYNC_USDC).transfer(address(airdrop), AMOUNT_TO_AIRDROP);
         vm.stopBroadcast();
     }
 }
```

This improves readability, removes duplicate literals, and makes deployment intent explicit.