The Rewarder Audit Report

Prepared by Ahmad Faraz

Auditing Protocol: `The Rewarder`

Date: August 04, 2025

Contents

1 Disclaimer

I, Ahmad Faraz, make every effort to understand the protocol and identify vulnerabilities within the given timeframe but holds no responsibility for missed issues. This audit is not an endorsement of the protocol's business logic or product and focuses solely on Solidity level vulnerabilities.

2 Risk Classification

2.1 Impact Table

Likelihood	**Critical**	**High**	**Medium**	**Low**
Critical	Critical	Critical	High	Medium
High	Critical	High	High/Medium	Medium
Medium	High	High/Medium	Medium	Medium/Low
Low	Medium	Medium	Medium/Low	Low

3 Audit Details

3.1 Scope

The audit covered the following files:

- `./src/`
- `TheRewarderDistributor.sol`

4 Protocol Summary

`TheRewarderDistributor` allows user to claim rewards, users must prove they're included in the chosen set of beneficiaries. The protocol has been optimized and allows claiming multiple tokens in the same transaction.

4.1 Roles

- **TheRewarderDistributor** Provides rewards to the chosen set of beneficiaries.

5 Executive Summary

The Rewarder smart contracts protocol audited by Ahmad Faraz. The audit identified 4 issues in total, classified as follows.

Severity Level	Issue Count
Critical	1
High	1
Gas	1
Informational	1
Total	**4**

6 Findings

6.1 Critical Severity

6.1.1 [C-1] Reclaim Rewards For Same Proof In `TheRewarderDistributor` Protocol

- **Description:** The `TheRewarderDistributor` protocol allows users to claim rewards for multiple tokens in a single transaction by submitting valid Merkle proofs. However, the `claimRewards` function defers marking claims as "used" until after all claims for a particular token have been processed. As a result, if a user includes the same claim multiple times in a single transaction, the system will process and approve each one, accumulating the reward amount, before checking whether the claims have already been made.

  This enables a malicious user to repeat the exact same claim object multiple times in a single call and receive rewards for each instance, draining the reward pool.

```
function claimRewards(Claim[] memory inputClaims, IERC20[] memory
    inputTokens) external {
        Claim memory inputClaim;
        IERC20 token;
        uint256 bitsSet;
        uint256 amount;

        for (uint256 i = 0; i < inputClaims.length; i++) {
            inputClaim = inputClaims[i];

            uint256 wordPosition = inputClaim.batchNumber / 256;
            uint256 bitPosition = inputClaim.batchNumber % 256;

            if (token != inputTokens[inputClaim.tokenIndex]) {
                if (address(token) != address(0)) {
                    if (!_setClaimed(token, amount, wordPosition,
                        bitsSet))
                        revert AlreadyClaimed();
                }

                token = inputTokens[inputClaim.tokenIndex];

                bitsSet = 1 << bitPosition;
                amount = inputClaim.amount;
            } else {
                bitsSet = bitsSet | (1 << bitPosition);
                amount += inputClaim.amount;
            }


            if (i == inputClaims.length - 1) {
                if (!_setClaimed(token, amount, wordPosition,
                    bitsSet))
                     revert AlreadyClaimed();
            }

            bytes32 leaf = keccak256(
                abi.encodePacked(msg.sender, inputClaim.amount)
            );

            bytes32 root = distributions[token].roots[inputClaim.
                batchNumber];

            if (!MerkleProof.verify(inputClaim.proof, root, leaf))
                revert InvalidProof();

            inputTokens[inputClaim.tokenIndex].transfer(
                msg.sender,
                inputClaim.amount
            );
        }
    }
```

- **Impact:** An attacker can reuse valid proofs to claim the same rewards multiple times, potentially draining the entire reward pool.
- **Cause:** Claims are grouped by token and only marked as claimed once per group. This allows duplicate claims within the same group to be processed and rewarded before any claim-tracking check is enforced.
- **Recommended Mitigation:** Ensure that each claim is validated and marked as claimed individually before any reward is transferred. Specifically:
  1. Call `_setClaimed` for each Claim, not only once after accumulating claims per token.
  2. Alternatively, pre-check for duplicate `batchNumbers` within the `inputClaims` array before proceeding.
  3. Update the logic to prevent processing multiple claims with the same `(msg.sender, batchNumber)` within a single transaction.
- **Proof of Concept:**
  1. A user submits a claim with valid Merkle proof and receives the reward.
  2. The same claim is submitted again with the same proof.
  3. Since the claim tracking logic does not persist the claim status correctly across calls, the reward is granted again.
  4. This process can be repeated until all protocol funds are exhausted.

  Run the code in `TheRewarderDistributor.t.sol`:

```
function testClaimReward() public {
        IERC20[] memory tokens = new IERC20[](2);
        Claim[] memory claim = new Claim[](3);

        claim[0] = Claim({
            batchNumber: 0,
            amount: 100,
            tokenIndex: 0,
            proof: firstUserProof
        });

        claim[1] = claim[0];
        claim[2] = claim[0];

        tokens[0] = usdcMock;

        vm.startPrank(0x0000000000000000000000000000000000000001);
        rewarderDistributor.claimRewards(claim, tokens);
        vm.stopPrank();
    }
```

6.2 High Severity

6.2.1 [H-1] Lack of Access Control on `TheRewarderDistributor::clean`

- **Description:** The `clean` function is used to sweep unallocated tokens from the `TheRewarderDistributor` once all rewards for a token have been claimed. However, this function lacks any form of access control, allowing any external caller to execute it.

```
function clean(IERC20[] calldata tokens) external {
        for (uint256 i = 0; i < tokens.length; i++) {
            IERC20 token = tokens[i];
            if (distributions[token].remaining == 0) {
                token.transfer(owner, token.balanceOf(address(this)
                    ));
            }
        }
    }
```

- **Impact:** An attacker can prematurely sweep tokens that are still meant to be distributed, effectively disrupting the reward system or stealing unclaimed token.

- **Cause:** Missing access control modifier allows any address to call privileged function..

- **Recommended Mitigation:** `onlyOwner` should be set properly to the actual owner of the protocol.

```
+ modifier onlyOwner() {
+     require(msg.sender == owner, "Not authorized");
+     _;
+ }

- function clean(IERC20[] calldata tokens) external  {
+ function clean(IERC20[] calldata tokens) external onlyOwner {
        for (uint256 i = 0; i < tokens.length; i++) {
            IERC20 token = tokens[i];
            if (distributions[token].remaining == 0) {
                token.transfer(owner, token.balanceOf(address(this)
                    ));
            }
        }
}
```

- **Proof of Concept:**
  Run the code in `TheRewarderDistributor.t.sol`:

```
function testCleanTokensWithoutAccessControl() public {
        IERC20[] memory tokens = new IERC20[](2);
        address attacker = address(0xBEEF);

        tokens[0] = usdcMock;
        tokens[1] = wethMock;

        uint256 balanceBefore = usdcMock.balanceOf(owner);

        vm.startPrank(attacker);
        rewarderDistributor.clean(tokens);
        vm.stopPrank();

        uint256 balanceAfter = usdcMock.balanceOf(owner);

        assertGt(
            balanceAfter,
            balanceBefore,
            "clean() should have transferred tokens"
        );
    }
```

6.3 Gas Severity

6.3.1 [G-1] Inefficient Access to `tokens.length` in Loop

- **Description:** In the `clean` function, `tokens.length` is read from `storage` on every iteration. This is slightly more expensive than caching it once in `memory`.

```
function clean(IERC20[] calldata tokens) external {

@>       for (uint256 i = 0; i < tokens.length; i++) {
            IERC20 token = tokens[i];
            if (distributions[token].remaining == 0) {
                token.transfer(owner, token.balanceOf(address(this)
                    ));
            }
        }

    }
```

- **Impact:** Unnecessary gas costs are incurred per loop iteration.
- **Recommended Mitigation:** Store the tokens length in a local memory variable.

```
+ modifier onlyOwner() {
+    require(msg.sender == owner, "Not authorized");
+     _;
+ }

- function clean(IERC20[] calldata tokens) external  {
+ function clean(IERC20[] calldata tokens) external onlyOwner {
        for (uint256 i = 0; i < tokens.length; i++) {
            IERC20 token = tokens[i];
            if (distributions[token].remaining == 0) {
                token.transfer(owner, token.balanceOf(address(this)
                   ));
            }
        }
}
```

6.4 Informational Severity

6.4.1 [I-1] Magic Number 256 Used in claimRewards

- **Description:** In the `claimRewards` function, the number '256 is used directly to calculate both the word position and bit position for Merkle claim tracking:

```
@> uint256 wordPosition = inputClaim.batchNumber / 256;
@> uint256 bitPosition = inputClaim.batchNumber % 256;
```

  While this is technically correct (each `uint256` word can track `256 bits`), using a hardcoded literal here reduces readability and makes future updates or audits harder.

- **Impact:** Reduces code clarity, makes intent less obvious to readers unfamiliar with bitmaps and increases risk of inconsistencies if the base size ever needs to change.

- **Recommended Mitigation:** Replace the magic number 256 with a named constant like `BITS_PER_WORD`:

```
+ uint256 constant BITS_PER_WORD = 256;
```

```
- uint256 wordPosition = inputClaim.batchNumber / 256;
- uint256 bitPosition = inputClaim.batchNumber % 256;

+ uint256 wordPosition = inputClaim.batchNumber / BITS_PER_WORD;
+ uint256 bitPosition = inputClaim.batchNumber % BITS_PER_WORD;
```

The End