Snowman Merkle Airdrop Audit Report

Joe LeFever (Sicher Height)

June 19, 2025

Joe LeFever - Sicher Height

Snowman Merkle Airdrop Audit Report

Version 1.0

Independent Security Audit

June 19, 2025

Snowman Merkle Airdrop Audit Report

Joe LeFever (Sicher Height)

June 19, 2025

Prepared by: Joe LeFever - Sicher Height Lead Auditor: - Joe LeFever

Table of Contents

Protocol Summary

The Snowman Merkle Airdrop protocol is a token distribution system that combines ERC20 tokens (Snow), ERC721 NFTs (Snowman), and Merkle tree-based airdrops. Users can earn or purchase Snow tokens, which can then be staked in the SnowmanAirdrop contract to receive proportional Snowman NFTs. The protocol features a 12-week farming period where users can earn free Snow tokens weekly or purchase them with ETH/WETH. The airdrop system utilizes Merkle proofs for efficient distribution and supports signature-based claiming on behalf of others.

Key Components: - **Snow.sol**: ERC20 token with buy/earn mechanics and fee collection - **Snowman.sol**: ERC721 NFT with on-chain Base64 encoding - **SnowmanAirdrop.sol**: Merkle tree-based airdrop system with signature verification

Disclaimer

The Joe LeFever - Sicher Height team 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 audit by the team 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Date: June 19, 2025
Auditor: Joe LeFever (Sicher Height)
Classification: Competitive Audit
Duration: 2 days

Scope

The following contracts were in scope for this audit:

```
src/
  Snow.sol
  Snowman.sol
  SnowmanAirdrop.sol
```

Lines of Code: ~300 total **Solidity Version:** ^0.8.24

Roles

- **Owner**: Deployer of Snow contract with administrative privileges
- **Collector**: Address authorized to collect fees from Snow token purchases
- **Users**: Can buy/earn Snow tokens and claim Snowman NFTs
- **Recipients**: Addresses eligible for airdrops via Merkle proofs

Executive Summary

This audit identified **6 High severity**, **6 Medium severity**, and **3 Low severity** vulnerabilities across the three contracts in the Snowman Merkle Airdrop protocol. The most critical issues involve missing access controls that allow unlimited NFT minting, fund theft through arbitrary token transfers, and broken signature verification due to a typo in the EIP-712 hash.

The protocol's core functionality is severely compromised by these vulnerabilities, particularly the unrestricted minting capability that completely breaks the tokenomics model. Immediate remediation of High severity findings is strongly recommended before any mainnet deployment.

Issues found

Severity	Number of issues found
High	6
Medium	6
Low	3
Total	**15**

Findings

[H-1] Global timer for Snow.earnSnow() prevents multiple users from earning Snow tokens (Access Control + DoS)

Description: The `earnSnow()` function uses a single global timer `s_earnTimer` that affects all users. When any user calls `earnSnow()`, it prevents all other users from calling the function for one week. This breaks the intended functionality where each user should be able to earn one Snow token per week independently.

Impact: Only one user per week can earn free Snow tokens across the entire protocol, preventing all other users from accessing this core feature and enabling griefing attacks.RetryClaude can make mistakes. Please double-check responses.

Proof of Concept:

```
// Day 0: Alice calls earnSnow()
function earnSnow() external canFarmSnow {
    // s_earnTimer = 0, so check passes
    _mint(msg.sender, 1); // Alice gets 1 token
    s_earnTimer = block.timestamp; // Global timer set
}

// Day 1: Bob calls earnSnow()
function earnSnow() external canFarmSnow {
```

```
    if (s_earnTimer != 0 && block.timestamp < (s_earnTimer + 1 weeks)) {
        revert S__Timer(); //  Bob's call reverts
    }
    // Bob gets nothing, has to wait because of Alice's timer
}
```

Recommended Mitigation: Replace the global timer with a per-user mapping

[H-2] Users can lose ETH when attempting to buy Snow.buySnow() with incorrect payment amounts (Logic Error + Fund Loss)

Description: The `buySnow()` function has flawed payment logic that can cause users to lose their ETH. If a user sends ETH but not the exact required amount, the function keeps their ETH and then attempts to charge them again via WETH transfer. This can result in double charging or fund loss when the WETH transfer fails.

Impact: Users lose ETH permanently when sending incorrect amounts, face double charging (paying both ETH and WETH), and can have funds trapped in the contract due to failed WETH transfers.

Proof of Concept:

```
// Scenario: s_buyFee = 1 ether, user wants 1 Snow token
function buySnow(1) external payable { // User sends 0.5 ETH by mistake
    if (msg.value == (s_buyFee * 1)) { // 0.5 ether != 1 ether
        // This branch skipped
    } else {
        // Contract keeps the 0.5 ETH the user sent
        i_weth.safeTransferFrom(msg.sender, address(this), 1 ether);
        // Also takes 1 ETH worth of WETH from user
        _mint(msg.sender, 1);
        // User paid 1.5 ETH total (0.5 ETH + 1 WETH) for 1 token
    }
}
```

Recommended Mitigation: Implement proper payment method validation and refund excess ETH.

[H-3] No Access Control on Snowman.mintSnowman() Allows Unlimited NFT Minting (Missing Access Control + Fund/Protocol Breaking)

Description: The `mintSnowman()` function in Snowman.sol has no access control restrictions, allowing any user to mint unlimited Snowman NFTs to any address. This completely breaks the intended tokenomics where NFTs should only be minted through the SnowmanAirdrop contract when users stake Snow tokens.

Impact: Complete protocol failure. Attackers can mint unlimited NFTs, making them worthless and breaking the staking/airdrop mechanism. Total loss of protocol integrity.

Proof of Concept:

```
function test_UnlimitedMinting() public {
   address attacker = makeAddr("attacker");
   vm.prank(attacker);
   snowman.mintSnowman(attacker, 1000000); // Mint 1M NFTs for free
   assertEq(snowman.balanceOf(attacker), 1000000);
}
```

Recommended Mitigation Add access control to ensure only the SnowmanAirdrop contract can mint:

[H-4] Arbitrary transferFrom in SnowmanAirdrop Allows Token Theft (Input Validation + Fund Drain)

Description: The `claimSnowman()` function uses user-controlled `receiver` parameter in `safeTransferFrom(receiver, address(this), amount)`, allowing attackers to drain any user's Snow tokens by setting `receiver` to victim addresses.

Impact: Direct fund theft. Attackers can steal Snow tokens from any user who has approved the airdrop contract or has sufficient balance.

Proof of Concept:

```
function test_ArbitraryTransferFrom() public {
    // Alice has 1 Snow token and approves airdrop
    vm.prank(alice);
    snow.approve(address(airdrop), 1);

    // Attacker steals Alice's tokens by using her as receiver
    bytes32 digest = airdrop.getMessageHash(alice);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(alKey, digest);

    vm.prank(satoshi); // Attacker transaction
    airdrop.claimSnowman(alice, AL_PROOF, v, r, s);

    // Alice's tokens stolen, she gets NFT but lost tokens
    assertEq(snow.balanceOf(alice), 0);
    assertEq(snow.balanceOf(address(airdrop)), 1);
}
```

Recommended Mitigation: Enforce that only `msg.sender` can transfer their own tokens

[H-5] MESSAGE_TYPEHASH Typo Breaks All Signature Verification (Typo + Signature Bypass)

Description: The `MESSAGE_TYPEHASH` constant contains a typo: `"SnowmanClaim(addres receiver, uint256 amount)"` is missing the 's' in "address". This causes all EIP-712 signature verification to fail as the hash won't match user-signed messages.

Impact: Complete airdrop system failure. No user can successfully claim through signature verification, making the entire signature-based claiming mechanism non-functional.

Recommended Mitigation: Fix the typo -

```
bytes32 private constant MESSAGE_TYPEHASH = keccak256("SnowmanClaim(address receiver, uint25
```

[H-6] Reentrancy in Snowman.mintSnowman() Allows Extra NFT Minting (Reentrancy + Token Inflation)

Description: The `mintSnowman()` function is vulnerable to reentrancy via the `_safeMint()` callback to `onERC721Received()`. State variables are updated after the external call, allowing malicious contracts to reenter and mint additional NFTs.

Impact: Attackers can mint more NFTs than intended, diluting the NFT supply and breaking the 1:1 Snow token to NFT ratio.

Proof of Concept:

```
// PoC - (Need malicious contract with onERC721Received)
contract MaliciousReceiver {
    function onERC721Received(...) external returns (bytes4) {
        if (attackCount < 3) {
            snowman.mintSnowman(address(this), 2); // Reenter
            attackCount++;
        }
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

Recommended Mitigation: Use ReentrancyGuard or follow Checks-Effects-Interactions pattern.

[M-1] Re-entrancy Snow.collectFee() vulnerability in fee collection allows potential exploitation (Re-entrancy)

Description: The `collectFee()` function uses a low-level call to transfer ETH to the collector without re-entrancy protection. This gives control to the collector contract during execution, potentially allowing re-entrancy attacks if the collector is a malicious contract.

Impact: Potential for re-entrancy attacks if collector is a malicious contract.

Proof of Concept:

```
// If collector is a malicious contract, it can re-enter during ETH transfer:
// collectFee() → ETH sent to collector → collector calls collectFee() again
```

Recommended Mitigation: Implement re-entrancy protection and follow checks-effects-interactions pattern.

[M-2] Transaction atomicity Snow.collectFee() issue causes fee collection failures (State Management)

Description: In `collectFee()`, if the WETH transfer succeeds but the ETH transfer fails, the entire transaction reverts. This can lead to situations where fee collection repeatedly fails due to ETH transfer issues, even when WETH transfer would succeed.

Impact: Fee collection can be permanently blocked if ETH transfer consistently fails.

Proof of Concept:

```
// If WETH transfer succeeds but ETH transfer fails (collector rejects ETH):
// WETH gets rolled back due to require() failure, no fees collected at all
```

Recommended Mitigation: Handle WETH and ETH transfers independently.

[M-3] No Double-Claim Protection in SnowmanAirdrop (Missing Validation + Double Spending)

Description:
The `claimSnowman()` function sets `s_hasClaimedSnowman[receiver] = true` but never checks this mapping before processing claims, allowing users to potentially claim multiple times.

Impact: Users could claim NFTs multiple times if they can generate multiple valid signatures, leading to unfair distribution and NFT inflation.

Proof of Concept:

```
function test_DoubleClaim() public {
    // Alice claims successfully
    vm.prank(alice);
    snow.approve(address(airdrop), 1);

    bytes32 digest = airdrop.getMessageHash(alice);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(alKey, digest);

    vm.prank(satoshi);
```

```
    airdrop.claimSnowman(alice, AL_PROOF, v, r, s);

    // Vulnerability: No check for s_hasClaimedSnowman[receiver] before processing
    // Function sets the flag but never checks it first
    assertTrue(airdrop.getClaimStatus(alice)); // Already claimed but no protection
}
```

Recommended Mitigation: Add double-claim check

[M-4] DoS via Unbounded Loop in Snowman.mintSnowman() (Resource Exhaustion + DoS)

Description: The `mintSnowman()` function contains a user-controlled loop with external calls (`_safeMint`), allowing attackers to cause out-of-gas errors by providing large `amount` values.

Impact: Function becomes unusable with large inputs, preventing legitimate minting operations and potentially locking the contract.

Proof of Concept:

```
// PoC
function test_DoSAttack() public {
    vm.expectRevert(); // Expect out of gas
    snowman.mintSnowman(address(this), type(uint256).max);
}
```

Recommended Mitigation: Add reasonable limits

[M-5] Unchecked Transfer Return Value in Snow.collectFee() (Unchecked Return + Silent Failure)

Description: The `collectFee()` function ignores the return value of `i_weth.transfer()`, causing silent failures if the WETH transfer fails while continuing to execute the rest of the function.

Impact: Collected fees could be permanently lost if WETH transfer fails, resulting in financial loss for the protocol.

Recommended Mitigation: Check return value or use SafeERC20

[M-6] Missing Merkle Root Validation in Constructor (Input Validation + Deployment Risk)

Description: The SnowmanAirdrop constructor accepts `_merkleRoot` parameter without validating it's not zero, potentially allowing deployment with an invalid merkle root that would make all proofs fail.

Impact: If deployed with zero merkle root, the entire airdrop system becomes non-functional as all merkle proofs would be invalid.

Recommended Mitigation: Add validation in constructor

[L-1] Incorrect Event Emission in Snowman.mintSnowman() (Logic Error + Integration Issues)

Description: The `SnowmanMinted` event parameter is named `numberOfSnowman` (suggesting count) but emits `s_TokenCounter` (token ID), creating semantic confusion and breaking off-chain integrations expecting count data.

Impact: Broken off-chain integrations, misleading analytics, and poor user experience as dApps expecting mint counts receive token IDs instead.

Recommended Mitigation: Either emit count or rename parameter.

[L-2] Missing event emission in Snow.earnSnow() function reduces transparency (Event Emission)

Description: The `earnSnow()` function does not emit an event when tokens are earned, unlike `buySnow()` which emits a `SnowBought` event. This inconsistency makes it difficult to track earning activities off-chain.

Impact: Reduced transparency and auditability.

Recommended Mitigation: Add event emission to earnSnow().

[L-3] Snow.sol Contract can collect accidental ETH transfers as fees (Unintended Fund Collection)

Description: The `collectFee()` function transfers all ETH in the contract to the collector using `address(this).balance`, including any ETH accidentally sent directly to the contract address. This ETH was not paid as fees for Snow tokens but gets collected as if it were.

Impact: Users who accidentally send ETH to the contract lose their funds and collector receives funds that weren't legitimate fees.

Recommended Mitigation: Track legitimate fees separately from accidental transfers.