Snowman Merkle Airdrop Security Audit

Prepared by Alade Jamiu

January 7, 2026

Snowman Merkle Airdrop Security Audit

AJTECH

January 7, 2026

Snowman Merkle Airdrop Security Audit

Prepared by: Alade Jamiu

- Alade Jamiu

Table of Contents

About Alade Jamiu

I am a security researcher dedicated to building robust and secure decentralized systems. With a focus on all smart contract languages, I work to identify and mitigate vulnerabilities in smart contracts to protect users and protocols.

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit 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 smart contract implementation.

Risk Classification

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

Severity Definitions

- **High**: Serious security vulnerability that can lead to loss of funds, protocol failure, or permanent disruption.
- **Medium**: Vulnerability that can affect protocol operation or user experience, but requires specific conditions to exploit.
- **Low**: Minor issue or potential improvement that doesn't pose an immediate threat but should be addressed for best practices.
- **Informational**: Observations regarding code quality, readability, or optimization that do not impact security.

Audit Details

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

```
d5f8e6a7b3c4d2e1f0a9b8c7d6e5f4a3b2c1d0e9
```

Scope

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

Protocol Summary

Snowman Merkle Airdrop is a protocol designed to distribute Snow tokens and Snowman NFTs to eligible users using a Merkle proof verification system. It consists of the airdrop contract, the Snow token (ERC20), and the Snowman NFT (ERC721).

Roles

- **Owner**: Admin with control over specific contract functions.
- **Receiver**: Users eligible to claim airdrop tokens and NFTs.
- **Minter**: Authorized addresses that can mint tokens/NFTs.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	3
Info	0
Total	**9**

Findings

High

[H-1] Unrestricted NFT Minting in `Snowman.sol`

Description * `mintSnowman` lacks any access control mechanisms or authorization checks. * Any external address can call this function to mint unlimited NFTs to any recipient.

```
function mintSnowman(address receiver, uint256 amount) external {
@>  // NO ACCESS CONTROL - Publicly accessible
    for (uint256 i = 0; i < amount; i++) {
        _safeMint(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

Risk Likelihood: discovery is inevitable via automated scanning/MEV bots. **Impact**: Complete destruction of NFT scarcity and tokenomics via unlimited supply injection.

Proof of Concept 1. Attacker calls `mintSnowman(attacker, 1_000_000)`. 2. Contract successfully mints 1M NFTs without requiring airdrop eligibility or owner approval. 3. Attacker dumps NFTs or holds majority supply.

Recommended Mitigation Add `onlyOwner` or authorized minter access control.

```
- function mintSnowman(address receiver, uint256 amount) external {
```

```
2 + function mintSnowman(address receiver, uint256 amount) external
      onlyOwner {
```

[H-2] Potential Reentrancy in `SnowmanAirdrop.claimSnowman`

Description * Function interacts with external token contract before updating internal claim state. * This permits reentrancy via malicious token hooks during `safeTransferFrom`.

```
1 @> i_snow.safeTransferFrom(receiver, address(this), amount); //
      Interaction
2    s_hasClaimedSnowman[receiver] = true; // Effect
```

Risk Likelihood: Malicious token contract or hook triggers during `transferFrom`. **Impact**: Attacker recursively claims Snowman NFTs, draining the entire supply.

Proof of Concept 1. Attacker uses a token with `tokensReceived` hook. 2. Attacker calls `claimSnowman`. 3. Hook re-enters `claimSnowman` before mapping is set to **`true`**. 4. Recursive calls continue until supply is drained.

Recommended Mitigation Update state before external interactions (CEI).

[H-3] Inconsistent `MESSAGE_TYPEHASH` in `SnowmanAirdrop`

Description * `MESSAGE_TYPEHASH` has a typo (“addres” instead of “address”), causing it to differ from standard EIP-712 off-chain hashing.

```
1 @> bytes32 private constant MESSAGE_TYPEHASH = keccak256("SnowmanClaim(
      addres receiver, uint256 amount)");
```

Risk Likelihood: Frontend tools hash using correct Solidity types (“address”). **Impact**: Signature recovery consistently fails in the contract, rendering `claimSnowman` unusable for all users.

Proof of Concept 1. Frontend generates `structHash` using `"address"`. 2. Contract checks signature against `structHash` using `"addres"`. 3. Hashes never match; `ecrecover` returns an address that doesn’t match the receiver.

Recommended Mitigation Correct the spelling in the type hash string.

Medium

[M-1] DoS to User via Front-running (Amount Balance)

Description * `claimSnowman` verifies signatures against the user's *current* balance rather than a fixed amount. * Attackers can forcefully send Snow tokens to a user to invalidate their pre-signed message.

```
@> uint256 amount = i_snow.balanceOf(receiver);
    // Signature must match this exact amount
```

Risk Likelihood: Attackers purchase Snow and front-run `claimSnowman` calls. **Impact**: User's signature becomes permanently invalid for the specific claim, causing DoS.

Proof of Concept 1. Alice signs a claim for 100 Snow. 2. Bob front-runs with a 1-wei transfer to Alice. 3. Alice now has 100.000...1 Snow. 4. Contract calculates hash for new balance; Alice's signature fails.

Recommended Mitigation Pass the `amount` as a parameter to `claimSnowman` and verify it against the Merkle proof/signature.

[M-2] Signature Malleability

Description * `_isValidSignature` lacks "low-s" enforcement, allowing non-canonical signature recovery.

Risk Likelihood: Attacker applies `n - s` transformation. **Impact**: Potential for replay attacks if specific signatures aren't correctly invalidated.

Recommended Mitigation Enforce `s <= n/2`.

[M-3] Unchecked Fee Overflow

Description * `s_buyFee * amount` can overflow in Solidity if not handled, leading to logic errors.

Recommended Mitigation Verify multiplication scaling or rely on 0.8+ reverts explicitly.

Low

[L-1] Missing Claim Status Check

Description * `claimSnowman` updates state at the end but doesn't check it at the start.

Recommended Mitigation Add `if (s_hasClaimedSnowman[receiver])revert`.

[L-2] Global Timer Reset DoS in `Snow.buySnow`

Description * `buySnow` resets a global `s_earnTimer`, blocking all free `earnSnow` claims for the entire protocol.

Risk Likelihood: Regular protocol usage or malicious micro-buys. **Impact**: Permanent suppression of free token claims for all users.

Recommended Mitigation Remove the timer reset from `buySnow` and use per-user mappings for `earnSnow`.

[L-3] Magic Number in Snow

- **Description**: Literal 1 used in minting.
- **Mitigation**: Use named constants.

Conclusion

The report identifies critical access control and logic flaws. Immediate integration of CEI and per-user state tracking is recommended.