Snowman Merkle Airdrop Security Review

Reviewer: Muhammad Faran **Date:** June 20, 2025 **Repo:** CodeHawks-Contests/2025-06-snowman-merkle-airdrop

---

Table of Contents

Disclaimer

I made 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 view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	**High**	**Medium**	**Low**
High	**High**	**High/Medium**	**Medium**
Medium	**High/Medium**	**Medium**	**Medium-Low**
Low	**Medium**	**Medium-Low**	**Low**

Audit Details

Scope:

```
src/
├── Snow.sol
├── Snowman.sol
└── SnowmanAirdrop.sol

script/
├── GenerateInput.s.sol
├── Helper.s.sol
├── SnowMerkle.s.sol
└── flakes/
    ├── input.json
    └── output.json
```

Protocol Summary

The Snowman Merkle Airdrop protocol enables users to acquire Snow tokens either by purchasing them with ETH/WETH or by farming (earning over time). Users can then claim Snowman NFTs by proving their eligibility via a Merkle proof and staking (transferring) their Snow tokens to the SnowmanAirdrop contract.

Roles

Owner

- The deployer of the contract (via `Ownable`).
- Has administrative privileges, such as transferring ownership.
- In the current contracts, the owner has limited direct actions.

Collector

- The address set as `s_collector` in `Snow.sol`.
- Responsible for collecting protocol fees (ETH/WETH) from the contract.
- Can also update the collector address.

User

Any participant who interacts with the protocol. Users can:

- Buy Snow tokens with ETH or WETH.
- Earn Snow tokens over time (farming).
- Claim Snowman NFTs by providing a valid Merkle proof and staking their Snow tokens.

Executive Summary

Severity	Number Of Issues Found
High	4
Medium	8
Low	1

Vulnerabilities Found

[H-1] Missing Claim Status Check allows Repeated NFT Minting

Scope

- `SnowmanAirdrop.sol`

Description

- Normally, airdrop contracts should prevent users from claiming more than once by enforcing a claim status check.

- The `SnowmanAirdrop` contract sets `s_hasClaimedSnowman[receiver] = true`, but **never checks it before minting**. This means users can repeatedly call `claimSnowman` and mint unlimited NFTs.

```
function claimSnowman(...) external nonReentrant {
    ...
    // @> s_hasClaimedSnowman is set
    s_hasClaimedSnowman[receiver] = true;

    i_snowman.mintSnowman(receiver, amount);
}
```

Risk

Likelihood:

- Occurs every time a user reuses the same valid Merkle proof and signature without being restricted by a claim status check.

Impact:

- Infinite minting of Snowman NFTs.
- Loss of trust and value in the NFT collection due to uncontrolled inflation.

Proof of Concept

```
// Repeated calls allow infinite NFT minting
for (uint i = 0; i < 10; i++) {
    airdrop.claimSnowman(receiver, proof, v, r, s);
}
```

Recommended Mitigation

Add a guard clause to prevent re-claiming.

```
function claimSnowman(...) external nonReentrant {
+   if (s_hasClaimedSnowman[receiver]) {
+       revert("Already claimed");
+   }

    ...
    s_hasClaimedSnowman[receiver] = true;
    ...
}
```

[H-2] Arbitrary From, allows Malicious user's to burn other User's Snow tokens

Scope

- `SnowmanAirdrop.sol`

Description

- Normally, `ERC20.transferFrom()` is used to transfer tokens **from the caller (`msg.sender`)** to a specified recipient. This respects the approval pattern where users only approve a spender to transfer *their own* tokens.

- In the current implementation, `i_snow.safeTransferFrom(receiver, address(this), amount)` allows **any caller to specify any address (`receiver`)** as the token source, potentially transferring tokens from arbitrary users **without consent**, assuming approval is in place.

```
// @audit arbitrary from used.
i_snow.safeTransferFrom(receiver, address(this), amount);
@>                     ^^^^^^^^
```

Risk

Likelihood:

- Any external address (including a malicious one) can call `claimSnowman()`.

- If a victim address (`receiver`) has approved the contract, the caller can **transfer that user's tokens** without authorization.

Impact:

- Users’ ERC20 (Snow) tokens can be **drained** by an attacker if they have approved the contract.

- Leads to **financial loss** and **trust erosion** in the protocol.

Recommended Mitigation

```diff
- i_snow.safeTransferFrom(receiver, address(this), amount);
+ i_snow.safeTransferFrom(msg.sender, address(this), amount);
```

Additionally:

- Enforce `require(msg.sender == receiver);` to ensure users can **only claim for themselves**, or explicitly restrict authorized relayers.

Here is the **Markdown-formatted vulnerability report** for the new finding in `Snow.sol`, specifically regarding the **unchecked transfer** on line 114:

[H-3] Unchecked Transfer Return Value leads to Potential Silent Failure in Fee Collection

Scope

- `Snow.sol`

Description

- Normally, when transferring ERC20 tokens using `transfer()` or `transferFrom()`, the return value (`bool success`) must be **checked** to ensure the operation succeeded.
- In the current `Snow.sol` contract, the `i_weth.transfer(s_collector, collection);` call **ignores the return value**.
- If a token (like some non-compliant ERC20s) returns `false` on failure **instead of reverting**, the function proceeds as if the transfer succeeded, leading to **silent fee loss**.

```
// Unchecked transfer return value
// line 114
i_weth.transfer(s_collector, collection);
@>     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Risk

Likelihood:

- This issue arises **when the WETH implementation is not strictly ERC20-compliant**, or if it is substituted with a similarly named token (malicious or otherwise) that returns `false` silently.
- Silent failure is hard to detect, especially in fee collection logic, which may be executed infrequently or in batch processes.

Impact:

- Fees may **not be successfully transferred**, resulting in **protocol revenue loss**.
- Users may assume the collector has received the fees, while funds remain trapped in the contract.

Proof of Concept

```
// Assume i_weth is a token that returns false instead of reverting
bool success = i_weth.transfer(s_collector, collection);
require(success, "WETH fee transfer failed");
```

Without this check, the protocol proceeds as if `transfer` succeeded, but the `s_collector` receives **nothing**.

Recommended Mitigation

```diff
- i_weth.transfer(s_collector, collection);
+ bool success = i_weth.transfer(s_collector, collection);
+ require(success, "WETH fee transfer failed");
```

Alternatively, use OpenZeppelin’s `SafeERC20` library to ensure safe handling of token transfers:

```diff
- i_weth.transfer(s_collector, collection);
+ i_weth.safeTransfer(s_collector, collection);
```

[H-4] Unsafe External Call in Loop causes Reentrancy Vulnerability

Scope

- `Snowman.sol`

Description

- The `mintSnowman()` function allows external addresses to receive NFTs via `_safeMint()` inside a loop.
- `_safeMint()` performs an external call to the `onERC721Received` function on the recipient contract **if the receiver is a contract**. This opens up the possibility for **reentrancy** if the receiver is a malicious contract.

```
function mintSnowman(address receiver, uint256 amount) external {
    for (uint256 i = 0; i < amount; i++) {
        @> _safeMint(receiver, s_TokenCounter); // unsafe external call
        emit SnowmanMinted(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

- Since `_safeMint()` is called **before** `s_TokenCounter++`, a reentrant call to `mintSnowman()` may cause **token ID reuse** or unpredictable state changes.

Risk

Likelihood:

- **Medium-High** — If `receiver` is a contract (like a malicious actor), it can reenter and exploit the minting logic.

Impact:

- **High** — Reentrancy could:
    - Reuse the same `s_TokenCounter` value,
    - Allow minting of unintended extra NFTs,
    - Cause infinite recursion and DoS,
    - Break token accounting or allow **free mints**.

Proof of Concept

```
contract ReentrantReceiver {
    Snowman public snowman;

    constructor(address _snowman) {
        snowman = Snowman(_snowman);
    }

    function onERC721Received(...) external returns (bytes4) {
        // Reenter during mint
        snowman.mintSnowman(address(this), 1);
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

- By calling `mintSnowman()` once, the attacker reenters during `onERC721Received()` to call it again, recursively.

Recommended Mitigation

```diff
+ bool internal locked;
+ modifier nonReentrant() {
+     require(!locked, "ReentrancyGuard: reentrant call");
+     locked = true;
+     _;
+     locked = false;
+ }

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

Or:

- Move state updates (like `s_TokenCounter++`) **before** any external call (such as `_safeMint()`), and
- Use OpenZeppelin’s `ReentrancyGuard`.

[M-1] Global `s_earnTimer` allow users to block each other from farmning Snow tokens

Scope

- `Snow.sol`

Description

- The `Snow` contract allows users to "earn" 1 Snow token every 1 week by calling `earnSnow()`. This feature is supposed to be individual to each user, enforcing a 1-week cooldown between claims.

- However, the `s_earnTimer` variable is implemented as a **single global timestamp**, not per-user. This means **all users share the same timer**, and one user's interaction affects everyone else. As a result, users can either block others from earning or front-run each other, depending on who calls the function first after the timer resets.

```
// earnSnow function
function earnSnow() external canFarmSnow {
    if (s_earnTimer != 0 && block.timestamp < (s_earnTimer + 1 weeks)) {
        revert S__Timer();
    }
    _mint(msg.sender, 1);
    s_earnTimer = block.timestamp;
}
```

Risk

Likelihood:

- Any time multiple users attempt to claim rewards using `earnSnow()`, they are subject to race conditions. This will **always** occur in production usage.

- A malicious user can grief others by calling `earnSnow()` repeatedly every week, effectively preventing others from earning tokens.

Impact:

- Users are unfairly blocked from accessing the rewards they are entitled to.

- Leads to centralization of token distribution and a poor user experience, possibly breaking economic assumptions of fairness.

Proof of Concept

```
// Assume Alice calls earnSnow(), sets the global timer
vm.prank(alice);
snow.earnSnow(); // works

// Immediately after, Bob tries to call earnSnow()
vm.prank(bob);
snow.earnSnow(); // fails due to s_earnTimer not being expired
```

Recommended Mitigation

- Use a mapping instead of a global variable ```diff
- uint256 private s_earnTimer;
- mapping(address => uint256) private s_earnTimer;

```diff
- if (s_earnTimer != 0 && block.timestamp < (s_earnTimer + 1 weeks)) {
+ if (s_earnTimer[msg.sender] != 0 && block.timestamp < (s_earnTimer[msg.sender] + 1
weeks)) {
```

```diff
    revert S__Timer();
}

- s_earnTimer = block.timestamp;
+ s_earnTimer[msg.sender] = block.timestamp;
```

[M-2] Eth trapped when sending incorrect msg.value hence user faces Eth losses

Scope

- `Snow.sol`

Description

- The `buySnow()` function in the `Snow` contract allows users to purchase Snow tokens using either **ETH** or **WETH**, based on whether the exact ETH amount matches the fee calculation.

- However, when the user mistakenly sends an incorrect amount of ETH (i.e., not exactly equal to `s_buyFee * amount`), the contract silently falls back to expecting WETH via `safeTransferFrom(...)`. In this fallback path, **any ETH sent with the transaction remains trapped in the contract**, as it is not refunded or used.

```
function buySnow(uint256 amount) external payable canFarmSnow {
    if (msg.value == (s_buyFee * amount)) {
        _mint(msg.sender, amount);
    } else {
        //@audit Mishandling of Eth, Eth trapped
@>         i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee * amount));
@>         _mint(msg.sender, amount);
        // s_earnTimer update...
    }
}
```

Risk

Likelihood:

- This will occur whenever a user sends the incorrect ETH amount — very common in DeFi protocols due to rounding errors, slippage miscalculations, or UI inconsistencies.

Impact:

- ETH is irreversibly locked in the contract without access control or withdrawal mechanism.

- User funds are silently lost with no error or refund, creating a poor UX and trust issue.

Proof of Concept

```
// User sends slightly more or less ETH than expected
uint256 amount = 1e18;
uint256 fee = snow.s_buyFee() * amount;

// Send slightly more ETH than required
snow.buySnow{value: fee + 1}(amount);

// ETH is not used or refunded
assert(address(snow).balance == 1);
```

Recommended Mitigation

- A recommended mitigation for the trapped ETH in buySnow() vulnerability is to reject transactions that send an incorrect ETH amount instead of silently falling back to WETH and trapping the ETH. This ensures users do not lose funds due to mistakes or UI errors. Stick to ETH only. ```diff function buySnow(uint256 amount) external payable canFarmSnow {
- if (msg.value == (s_buyFee * amount)) {
- _mint(msg.sender, amount);
- } else {
- //@audit Mishandling of Eth, Eth trapped
- i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee * amount));
- _mint(msg.sender, amount);
- // s_earnTimer update...
- }
- if (msg.value == (s_buyFee * amount)) {
- _mint(msg.sender, amount);
- } else {
- revert("Incorrect ETH amount sent");
- } }

  ```
  Optionally, you may also add:
  ```

```
// Add a refund mechanism if ETH is sent accidentally
if (msg.value > 0) {
    (bool success, ) = msg.sender.call{value: msg.value}("");
    require(success, "Refund failed");
}
```

[M-3] Incorrect `s_buyFee` scaling allows users getting overcharged on Minting Snow tokens

Scope

- `Snow.sol`

Description

- Normally, `s_buyFee` should represent the fee per Snow token **in wei**, e.g., `0.01 ether = 10**16`.
- However, in the `Snow` contract constructor, `s_buyFee` is mistakenly multiplied by `PRECISION` (10^{18}), leading to a fee that is **18 decimal places too large**. This causes the ETH requirement in `buySnow()` to be **unreasonably high**, resulting in failed transactions or fallback to WETH unintentionally.

```
constructor(address _weth, uint256 _buyFee, address _collector) ERC20("Snow", "S")
Ownable(msg.sender) {
    ...
    // @audit Mishandling of Eth, Poor Precision
@>  s_buyFee = _buyFee * PRECISION;
    ...
}
```

Risk

Likelihood:

- This issue **always occurs** if `_buyFee` is provided in wei (as is standard), and the developer multiplies it by `PRECISION`.
- It is highly likely to affect all deployments and user interactions unless specifically corrected.

Impact:

- Users will be required to send an unreasonably large amount of ETH (e.g., 10^{34} wei) to mint even 1 token.

Proof of Concept

```
// Deployer passes 0.01 ether as the _buyFee
uint256 _buyFee = 10**16; // 0.01 ether
// Contract sets s_buyFee = 10^16 * 10^18 = 10^34

// User tries to buy 1 Snow token by sending 0.01 ether
snow.buySnow{value: 10**16}(1);

// Fails: msg.value != s_buyFee * amount
```

Recommended Mitigation

```diff
- s_buyFee = _buyFee * PRECISION;
+ s_buyFee = _buyFee;
```

[M-4] Unverified input URI in Constructor allows Garbage NFT's to be created

Scope

- `Snowman.sol`

Description

- Normally, when setting a metadata URI (such as an SVG URI) for NFTs, the input should be validated to ensure it's non-empty and conforms to expected URI schemes. This helps prevent incorrect or malicious metadata configurations.
- In the constructor of the `Snowman` contract, the `s_SnowmanSvgUri` is directly assigned without validation. If an empty or malformed URI is passed, it will break the visual representation and metadata of all minted NFTs.

```
constructor(string memory _SnowmanSvgUri) ERC721("Snowman Airdrop", "SNOWMAN")
Ownable(msg.sender) {
    //@audit
@>  s_SnowmanSvgUri = _SnowmanSvgUri;
}
```

Risk

Likelihood:

- This will occur if the deployer mistakenly passes an empty string, whitespace, or malformed URI during deployment.

Impact:

- All NFTs minted under this contract will have invalid metadata rendering in wallets and marketplaces.
- Breaks NFT user experience and potentially affects floor price due to invisible/invalid NFTs.

Proof of Concept

```
// Deploy the Snowman contract with an empty URI
string memory emptyURI = "";
Snowman snowman = new Snowman(emptyURI);

// mint a token
snowman.mintSnowman(user, 1);

// tokenURI will now contain an empty image field
string uri = snowman.tokenURI(0);
// Resulting JSON metadata will be invalid for marketplaces
```

Recommended Mitigation

```diff
constructor(string memory _SnowmanSvgUri) ERC721("Snowman Airdrop", "SNOWMAN")
Ownable(msg.sender) {
-    s_SnowmanSvgUri = _SnowmanSvgUri;
+    require(bytes(_SnowmanSvgUri).length > 0, "Snowman: Empty URI not allowed");
+    s_SnowmanSvgUri = _SnowmanSvgUri;
}
```

[M-5] Improper Access Control on `mintSnowma` allows Unrestricted Minting

Scope

- `Snowman.sol`

Description

- Normally, a `mint` function in an NFT contract should include **access control** to restrict who is allowed to mint tokens. This ensures only trusted contracts or parties can issue new NFTs.
- In the current implementation of the `Snowman` contract, the `mintSnowman` function is marked `external` and has **no access control**, meaning **any external address** can mint unlimited NFTs.

```solidity
function mintSnowman(address receiver, uint256 amount) external {
@> // No access control – anyone can call this
    for (uint256 i = 0; i < amount; i++) {
        _safeMint(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

Risk

Likelihood:

- This will occur whenever a malicious actor directly calls `mintSnowman()` to mint unauthorized NFTs.

Impact:

- Anyone can arbitrarily inflate the Snowman NFT supply.
- Completely breaks the trust model and devalues legitimately claimed NFTs.

Proof of Concept

```
Snowman snowman = Snowman(SNOWMAN_ADDRESS);

// Malicious user calls mint directly
snowman.mintSnowman(msg.sender, 1000); // mints 1000 fake NFTs
```

Recommended Mitigation

Restrict access to `mintSnowman` so only the `SnowmanAirdrop` contract can call it. Use a one-time setter or constructor injection pattern. Also `i_airdrop` represents address of Snowman Airdrop contract so only via Snowman Airdrop contract Snowman NFT's can be minted.

```diff
+ address private immutable i_airdrop;

- constructor(string memory _SnowmanSvgUri) ERC721("Snowman Airdrop", "SNOWMAN") Ownable(msg.sender) {
+ constructor(string memory _SnowmanSvgUri, address airdrop) ERC721("Snowman Airdrop", "SNOWMAN") Ownable(msg.sender) {
+     require(airdrop != address(0), "Invalid airdrop address");
+     i_airdrop = airdrop;
      s_TokenCounter = 0;
      s_SnowmanSvgUri = _SnowmanSvgUri;
  }

function mintSnowman(address receiver, uint256 amount) external {
+   if (msg.sender != i_airdrop) {
+       revert SM__NotAllowed();
+   }

    for (uint256 i = 0; i < amount; i++) {
        _safeMint(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

This ensures only the trusted `SnowmanAirdrop` contract can call `mintSnowman()` while preventing any external abuse.

[M-6] Denial of Service in NFT Mint Loop and hence Minting Fails

Scope

`Snowman.sol`

Description

- Normally, minting functions should limit the number of iterations or allow controlled batching to avoid exceeding block gas limits.

- The current implementation of `mintSnowman` performs a **loop based on an arbitrary `amount`** without any upper bound or batching logic. This creates a **Denial of Service (DoS) risk** if a large `amount` is passed, as the transaction may exceed the block gas limit and always revert.

```
function mintSnowman(address receiver, uint256 amount) external {
    for (uint256 i = 0; i < amount; i++) {
@>     _safeMint(receiver, s_TokenCounter);
        emit SnowmanMinted(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

Risk

Likelihood:

- This occurs when a large `amount` is passed to `mintSnowman`.

Impact:

- Makes it impossible to mint NFTs in one transaction when `amount` is too large.
- Could halt onboarding or claiming process if not handled properly.

Proof of Concept

```
// Simulate a large mint causing out-of-gas
snowman.mintSnowman(attacker, 1_000_000); // Likely to exceed block gas limit and revert
```

Recommended Mitigation

Add a **reasonable upper limit** on the `amount` or enforce a **batching strategy**.

```
function mintSnowman(address receiver, uint256 amount) external {
+   if (amount > 10) {
+       revert("Too many NFTs in one mint");
+   }

    for (uint256 i = 0; i < amount; i++) {
        _safeMint(receiver, s_TokenCounter);
        emit SnowmanMinted(receiver, s_TokenCounter);
        s_TokenCounter++;
    }
}
```

This protects against denial-of-service due to gas exhaustion while still enabling bulk minting in controlled sizes.

[M-7] Use of Dynamic `balanceOf` in Proof Generation makes Claim become Invalid

Scope

- `SnowmanAirdrop.sol`

Description

- The contract uses `balanceOf(receiver)` to determine the amount of Snow tokens staked, which is later used in the Merkle proof and EIP712 signature.
- **Problem**: `balanceOf` is **dynamic**, and if the balance changes between proof generation and contract interaction, the proof becomes invalid — **introducing fragility and failure risk**.

```
if (i_snow.balanceOf(receiver) == 0) {
    revert SA__ZeroAmount();
}

uint256 amount = i_snow.balanceOf(receiver); // @> dynamic lookup
...
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(receiver, amount))));
```

Risk

Likelihood:

- Highly likely if users accidentally transfer Snow tokens after proof/signature is generated but before calling `claimSnowman`.

Impact:

- Users are unable to claim NFTs despite having valid proof/signatures at generation time.
- Poor user experience and potential loss of eligibility.

Proof of Concept

```
uint256 amount = 1000;
generateProof(receiver, amount);
snow.transfer(receiver, amount);

// user spends tokens before claiming
snow.transfer(other, 1000);

// claim fails due to changed balance
airdrop.claimSnowman(receiver, proof, v, r, s);
```

Recommended Mitigation

Do not depend on `balanceOf()` inside the contract. Instead, accept `amount` as a parameter signed via EIP712 and encoded into the Merkle tree.

```diff
function claimSnowman(address receiver, uint256 amount, bytes32[] calldata proof, ...)
external {
-   uint256 amount = i_snow.balanceOf(receiver); // remove this dynamic call
+   // Use amount passed in by the user that matches the signed message and Merkle
leaf
}
```

Here is the **Markdown-formatted vulnerability report** for the **strict equality check** in `Snow.sol` inside the `buySnow` function:

[M-8] Strict Equality on ETH Payment Always Falls into Fallback Logic

Scope

- `Snow.sol`

Description

- Normally, smart contracts should **not use strict equality checks** (`==`) when comparing `msg.value` to an expected ETH amount.
- In the `buySnow` function, a strict equality is used to compare the sent Ether (`msg.value`) to the computed price:

```
// Vulnerable strict equality on line 91
if (msg.value == (s_buyFee * amount)) {
@>   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    _mint(msg.sender, amount);
} else {
    i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee * amount));
    _mint(msg.sender, amount);
}
```

- **Floating point rounding issues**, **tiny dust ETH overpayments**, or **gas tokens** causing unexpected balances often make `msg.value == expectedAmount` unreliable. This leads the logic to always fall into the `else` block, which assumes WETH will be used — potentially trapping ETH.

Risk

Likelihood:

- **Very High** — small discrepancies are common in real-world usage (e.g., UI rounding errors, wallets sending a few wei extra), causing this check to fail frequently.

- Breaks UX and causes **unexpected fallback to WETH** behavior, especially when ETH was sent.

Impact:

- ETH sent by the user is **trapped**, and **WETH is incorrectly deducted** again from their wallet via `safeTransferFrom`.
- Potential loss of user funds and **broken purchasing flow**.

Proof of Concept

```
function buySnow(uint256 amount) external payable {
    // Let's say s_buyFee * amount = 1 ether
    // But user accidentally sends 1.000000000000000001 ether
    // The following will fail due to strict equality
    if (msg.value == (1 ether)) {
        // this won't run
    } else {
        // this will run, but WETH is expected even though ETH was already sent
    }
}
```

Recommended Mitigation

```diff
- if (msg.value == (s_buyFee * amount)) {
+ if (msg.value >= (s_buyFee * amount)) {
```

- Optionally refund excess ETH:

```
uint256 required = s_buyFee * amount;
require(msg.value >= required, "Insufficient ETH sent");
_mint(msg.sender, amount);

if (msg.value > required) {
    payable(msg.sender).transfer(msg.value - required);
}
```

[L-1] Typo in MESSAGE_TYPEHASH (addres) makes all Signature Verifications Fail

Scope

- `SnowmanAirdrop.sol`

Description

- The `MESSAGE_TYPEHASH` used for EIP712 encoding defines the struct as `"SnowmanClaim(addres receiver, uint256 amount)"`.
- This is a **typo**: `"addres"` is not a valid type, and this hash will **never match any signed messages**, resulting in all signature verifications failing.

```
// @> Typo: 'addres' instead of 'address'
bytes32 private constant MESSAGE_TYPEHASH = keccak256("SnowmanClaim(addres receiver,
uint256 amount)");
```

Risk

Likelihood:

- Always occurs — signature verification will fail 100% of the time.

Impact:

- No user will ever be able to claim their Snowman NFT using EIP712 signatures.
- Complete failure of signature-based verification.

Proof of Concept

The following solidity example proves that all attempts to verify EIP712 messages will fail with this bug.

```
// All attempts to verify valid EIP712 messages will fail
bool valid = _isValidSignature(receiver, digest, v, r, s); // always returns false
```

Recommended Mitigation

Fix the typo in the `MESSAGE_TYPEHASH`:

```diff
- bytes32 private constant MESSAGE_TYPEHASH = keccak256("SnowmanClaim(addres receiver,
uint256 amount)");
+ bytes32 private constant MESSAGE_TYPEHASH = keccak256("SnowmanClaim(address
receiver, uint256 amount)");
```