Snowman Merkle Airdrop Audit Report

Sagar Rana

June 17, 2025

Snowman Merkle Airdrop Audit Report

Sagar Rana

June 17, 2025

Prepared by: Sagar Rana

Lead Auditors: - Sagar Rana

Table of Contents

Protocol Summary

The protocol revolves around the Snow.sol, Snowman.sol, and SnowmanAirdrop.sol contracts. Snow.sol is an ERC20 token that allows holders to claim Snowman NFTs through a staking mechanism in the SnowmanAirdrop contract. Snow tokens can be earned weekly for free or purchased using WETH or ETH before the ::FARMING_DURATION expires. By staking Snow tokens, users receive Snowman NFTs proportional to their Snow token holdings. Snowman.sol is an ERC721 contract that stores data entirely on-chain using Base64 encoding. The SnowmanAirdrop contract employs Merkle trees for efficient NFT distribution, enabling recipients to claim Snowman NFTs directly or via someone else using cryptographic signatures. This protocol offers a seamless interaction between ERC20 and ERC721 assets through staking and airdrop functionalities.

Disclaimer

Sagar Rana 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

Commit hash: b63f391444e69240f176a14a577c78cb85e4cf71

Scope

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

```

```
script/
#-- GenerateInput.s.sol
#-- Helper.s.sol
#-- SnowMerkle.s.sol
#-- flakes/
    #-- input.json
    #-- output.json
```

Roles

- Owner: Owner of the `Snow` token, also the deployer
- Collector: Collector is the address llowed to withdraw all fees accumulated in the `Snow` contract
- User: User who will be able to buy and earn Snow tokens and exchange them for NFTs during airdrop

Issues found

Severity	Number of issues
High	5
Medium	6
Low	6
Informational	8
Gas	3
Total	28

Findings

High

[H-1] Buying Snow tokens resets the earning timer everytime and user has to wait for a week to earn everytime someone buys the token

Description

- Inside the `Snow::buySnow()` function, the `s_earnTimer` is updated to the current time at the end of the function.
- The s_earnTimer acts as a flag for keeping track of the time the last earni has been taken place.
- Updating the variable while buying the token will result in the reste of the timer, and now the user has to wait a week for earning a token
- This is not the expected behaviour, as users can buy tokens anytime and earn once a week.
- A "Denial of Service" can be performed by calling this function repeatedly every 1 week so that nobody is able to earn tokens

```
function buySnow(uint256 amount) external payable canFarmSnow {
    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);
    }
@>  s_earnTimer = block.timestamp;

    emit SnowBought(msg.sender, amount);
}
```

Risk Likelihood:

- Whenever someone buys the token

Impact:

- Nobody will be able to earn tokens

Proof of Concept Add the following test case to the test suite of `Snow`

```
function test_earnRevert() public {
    vm.startPrank(ashley);
    vm.warp(block.timestamp + 1 weeks);
    weth.mint(ashley, 5e18);
    weth.approve(address(snow), 5e18);
    snow.buySnow(1);
    vm.expectRevert();
    snow.earnSnow();
}
```

Recommended Mitigation

1. Remove the timer updation logic in `buySnow()`

```
-    s_earnTimer = block.timestamp;
```

[H-2] The `Snowman::mintSnowman()` function has no access checks and anyone can call it to mint an NFT for free

Description

- The `mintSnowman()` function is supposed to be called by the `SnowmanAirdrop` contract to mint the NFT to eligible wallets.
- However, the function has no access control to make sure that only the Airdrop function can call it
- As the function is right now, anyone can call the function as many times as they want and mint NFTs to their own wallets without even interacting with the Airdrop contract

```
@>   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**:

- Always
- There is no restriction in calling the function

Impact:

- NFTs can be minted for free as many times as the caller wants
- NFTs can be minted even without being part of the airdrop

Proof of Concept Add the following test case to the test suite of `Snowman`

```
function test_anyoneCanMint() public {
    nft.mintSnowman(alice, 100);
}
```

Recommended Mitigation There should be a variable that stores the address of the airdrop contract and an owner-only function to update the address of the airdrop. Then a modifier can be created and used to make sure that only the airdrop contract address can call the function

```
+    address public airdrop;

+    modifier onlyAirdrop() {
+        assert(msg.sender == airdrop);
+        _;
+    }

+    function changeAirdropAddress(address _airdrop) external onlyOwner
    {
+        airdrop = _airdrop;
+    }
```

[H-3] `SnowmanAirdrop::claimSnowman()` has no checks to ensure that an address has already claimed and a person can claim twice

Description

- Airdrop contract uses markle root to check for eligibility
- However, address remains eligible even if person has claimed once, since there are no other checks

```
function claimSnowman(address receiver, bytes32[] calldata merkleProof,
     uint8 v, bytes32 r, bytes32 s)
    external
    nonReentrant
{
    if (receiver == address(0)) {
        revert SA__ZeroAddress();
    }
    if (i_snow.balanceOf(receiver) == 0) {
        revert SA__ZeroAmount();
    }

    if (!_isValidSignature(receiver, getMessageHash(receiver), v, r, s)
       ) {
        revert SA__InvalidSignature();
    }

    uint256 amount = i_snow.balanceOf(receiver);

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

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

    i_snow.safeTransferFrom(receiver, address(this), amount); // send
       tokens to contract... akin to burning

    s_hasClaimedSnowman[receiver] = true;

    emit SnowmanClaimedSuccessfully(receiver, amount);

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

Risk Likelihood:

- very high

Impact:

- Users claiming more than once breaks invariants

Proof of Concept Add the following test case to the test suite of `SnowmanAirdrop`. The contract has a merkle hash where alice's wallet with 1 Snow token is eligible for minting the airdrop. Therefore, we are minting snowman once. Then, Alice's wallet address becomes 0. However, we are transferring Alice another 1 Snow token and thus she again becomes eligible for the airdrop and can claim an NFT, as proved by the test

```
function test_claimAgain() public {
    assert(nft.balanceOf(alice) == 0);
    vm.prank(alice);
    snow.approve(address(airdrop), 2);

    bytes32 alDigest = airdrop.getMessageHash(alice);

    (uint8 alV, bytes32 alR, bytes32 alS) = vm.sign(alKey, alDigest);

    vm.prank(satoshi);
    airdrop.claimSnowman(alice, AL_PROOF, alV, alR, alS);

    assert(nft.balanceOf(alice) == 1);
    assert(nft.ownerOf(0) == alice);

    vm.prank(bob);
    snow.transfer(alice, 1);

    vm.prank(satoshi);
    airdrop.claimSnowman(alice, AL_PROOF, alV, alR, alS);

    assert(nft.balanceOf(alice) == 2);
    assert(nft.ownerOf(1) == alice);
}
```

Recommended Mitigation Add a mapping from address to bool which keeps track of the people who have claimed the token, and add a check using the mapping to prevent people from claiming again

```
+   mapping(address=>bool) public hasClaimed;
.
.
.
function claimSnowman(address receiver, bytes32[] calldata merkleProof,
     uint8 v, bytes32 r, bytes32 s)
    external
    nonReentrant
{
+   assert(hasClaimed[msg.sender] != true)
    if (receiver == address(0)) {
        revert SA__ZeroAddress();
    }
    if (i_snow.balanceOf(receiver) == 0) {
        revert SA__ZeroAmount();
    }

    if (!_isValidSignature(receiver, getMessageHash(receiver), v, r, s)
        ) {
        revert SA__InvalidSignature();
    }
+   hasClaimed[msg.sender] = true;
```

[H-4] Merkle root uses address and its balance to check for eligibility, users can become eligible after token transfers

Description

- Eligibility is checked by calculating the merkle root. The original merkle root has its leafs created from address and its balances
- If someone transfers tokens to or from some other address, they will become ineligible for the airdrop

```
uint256 amount = i_snow.balanceOf(receiver);

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

Risk Likelihood:

- Whenever tokens are tranferred out/in

Impact:

- User becomes ineligible for airdrop and can't claim NFT

Proof of Concept Add the following test case to the test suite of `SnowmanAirdrop` Here, Bob malaciously tranferred out his tokens to Alice, thus increasing her Snow balance, which makes her ineligible for airdrop and claimSnowman fails.

```
function test_becomeIneligible() public {
    assert(nft.balanceOf(alice) == 0);
    vm.prank(alice);
    snow.approve(address(airdrop), 2);

    bytes32 alDigest = airdrop.getMessageHash(alice);

    (uint8 alV, bytes32 alR, bytes32 alS) = vm.sign(alKey, alDigest);

    vm.prank(bob);
    snow.transfer(alice, 1);

    vm.startPrank(satoshi);
    vm.expectRevert();
    airdrop.claimSnowman(alice, AL_PROOF, alV, alR, alS);
}
```

Recommended Mitigation Only address should be used to check for eligibility

[H-5] `s_earnTimer` is not address specific so nobody will be able to earn Snow for a week after one address has called the `Snow::earnSnow()` function

Description

- The `Snow` contract uses a variable `s_earnTimer` to keep track of the last time a Snow was earned
- However, the variable is not address specific. So once `earnSnow()` has been called, the function will be uncallable by anyone for a week

```
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:

- Very high. Whenever someone calls the `earnSnow()` for the first time in a week

Impact:

- Nobody will be able to earn snow for a week after someone else has earned it once during the week

Proof of Concept Add the following test case to the test suite of `Snow`

```
function test_lockSnow() public {
    vm.prank(ashley);
    snow.earnSnow();
    vm.startPrank(jerry);
    vm.expectRevert();
    snow.earnSnow();
}
```

Recommended Mitigation Make s_earnTimer user specific by turning it into a mapping of address to uint256

```
-   uint256 private s_earnTimer;
+   mapping(address=>uint256) private s_earnTimer;
.
.
.
function earnSnow() external canFarmSnow {
    if (s_earnTimer[msg.sender] != 0 && block.timestamp < (s_earnTimer
        + 1 weeks)) {
        revert S__Timer();
    }
    _mint(msg.sender, 1);

    s_earnTimer[msg.sender] = block.timestamp;
}
```

Medium

[M-1] Weird ERC-20 tokens can cause unexpected behaviours

Description The `collectFee()` function collects all WETH tokens and native ETH held by the contract and sends them to the collector address. It assumes that the full balance of WETH can be successfully transferred to `s_collector` using the standard transfer() method.

However, the function does not account for weird, non-standard ERC20 tokens which may behave differently. These include: - Fee-on-Transfer Tokens which deduct a fee during transfer, - Rebase Tokens which automatically adjust balances, - ERC777 Tokens which can trigger hooks and reentrancy, - Blacklisting Tokens which may restrict transfers based on the sender/recipient.

```
function collectFee() external onlyCollector {
    uint256 collection = i_weth.balanceOf(address(this));

@>  i_weth.transfer(s_collector, collection);

    (bool collected,) = payable(s_collector).call{value: address(this).
       balance}("");
    require(collected, "Fee collection failed!!!");
}
```

Risk **Likelihood**:

- When the token address given as WETH is non-standard (e.g., a fee-on-transfer token or ERC777),
- When `s_collector` is blacklisted or recipient balance changes due to a rebase mid-transaction,

Impact:

- Loss of funds due to partial or failed transfers that go unnoticed,
- Reentrancy vulnerability via ERC777 hooks,
- Failed transactions due to transfer restrictions or changes in balance logic.

Proof of Concept

1. Fee-on-transfer token: Only `amount - fee` is received by `s_collector`, meaning `i_weth.balanceOf` is not equal to transferred amount. Silent fund loss.

```
function transfer(address recipient, uint256 amount) external returns (
    bool) {
    uint256 fee = amount / 100; // 1% fee
    _transfer(msg.sender, feeReceiver, fee);
    _transfer(msg.sender, recipient, amount - fee);
    return true;
}
```

2. Rebase token: Balances can change between the `balanceOf` call and the `transfer()` call. Can result in over/under sending.

```
function rebase(uint256 epoch, int256 supplyDelta) external {
    if (supplyDelta == 0) return;
    totalSupply = totalSupply + uint256(supplyDelta);
    for (address acc : holders) {
        balances[acc] += balances[acc] * supplyDelta / totalSupply;
    }
}
```

3. ERC777 token: Can re-enter into `collectFee()` if `s_collector` is a contract implementing `tokensReceived()`. Leads to reentrancy vulnerabilities.

```
function send(address recipient, uint256 amount, bytes memory data)
    external {
    _callTokensToSend(...); // Can call reentering hooks
    _move(msg.sender, recipient, amount);
    _callTokensReceived(...); // Reentrancy entry point
}
```

4. Blacklisting token: Transfer fails if `s_collector` is blacklisted. Results in permanent locking of funds in the contract.

```
function transfer(address recipient, uint256 amount) external returns (
    bool) {
    require(!blacklisted[msg.sender] && !blacklisted[recipient], "
        Blacklisted");
    _transfer(msg.sender, recipient, amount);
    return true;
}
```

Recommended Mitigation

```
- i_weth.transfer(s_collector, collection);
+ SafeERC20.safeTransfer(i_weth, s_collector, i_weth.balanceOf(address(
    this)));
```

Also consider implementing reentrancy guards around ETH and token transfers to further harden the contract against ERC777-based attacks.

[M-2] Malicious WETH address in `Snow` can allow reentrancy attacks

Description

- The `Snow::buySnow()` calls external functions from `i_weth` contract. The `safeTransferFrom()` function is a function from OpenZeppelin's SafeErc20 interface and internally it calls the `transferFrom()` function of the ERC20 token contract.
- A malicious contract inheriting from ERC20 contract but overriding the `transferFrom()` function can contain attacker's own logic and cause reentrancy vulnerability

```
@>  i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
    amount));
```

Risk **Likelihood**:

- Deployer of contract puts malicious contract address either deliberately or due to human errors

Impact:

- User can call `buySnow()` once with the amount needed for buying, but then the contract will reenter the contract and transfer as many tokens as needed

Proof of Concept Add the following testcase to the `Snow.t.sol` test suite:

```
function test_buyReentrant() public {
    vm.startPrank(ashley);
    TestWETH test_weth = new TestWETH();
    uint256 fee = 1;
    Snow newSnow = new Snow(address(test_weth), fee, collector);
    newSnow.buySnow(10);
    uint256 res = newSnow.balanceOf(ashley);
    assertEq(res, 100);
}
```

Also add the following contract to the import list. This is the malicious reentrant contract:

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.24;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Snow} from "../../src/Snow.sol";

contract TestWETH is ERC20 {

    uint256 public counter;
    address public immutable owner;
    address public snow_addr;

    constructor() ERC20("MockWETH", "mWETH") {
        owner = msg.sender;
    }

    function transferFrom(address, address, uint256) public override
       returns (bool) {
        Snow snow = Snow(msg.sender);
        counter++;
        if (counter < 10) {
            snow.buySnow(10);
        }
        snow.transfer(owner, snow.balanceOf(address(this)));
        return true;
    }
}
```

Recommended Mitigation Add reentrancy guards to the function. OpenZeppelin's reentrancy guard works by locking the function once it has been called, till the time the call has been fully executed. This means that once entered, nobody will be able to reenter the contract due to the locking mechanism

```
+import {ReentrancyGuard} from "@openzeppelin/contracts/utils/
   ReentrancyGuard.sol";

-contract Snow is ERC20, Ownable {
+contract Snow is ERC20, Ownable, ReentrancyGuard {
.
.
.
-    function buySnow(uint256 amount) external payable canFarmSnow {
+    function buySnow(uint256 amount) external payable canFarmSnow
   nonReentrant {
```

[M-3] `Snow::buySnow()` not checking for return value from token transfer

Description

- In `Snow::buySnow()` we are transferring tokens from caller to the contract. SafeERC20's `safeTransferFrom()` function is being used to do the operation
- `safeTransferFrom()` internally calls the `transferFrom()` function. Even if the tx fails and returns false, the function does not check for the result and continues the operation. The transaction may be forced to fail deliberately so that the caller can get Snow tokens

for free

```
@>  i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
   amount));
    _mint(msg.sender, amount);
```

Risk **Likelihood**:

- Transaction fails
- Malicious WETH contract
- Deliberate tx fails

Impact:

- Snow tokens can be bought for free

Proof of Concept Add the the following testcase to the test suite of `Snow`:

```
function test_buyFreeToken() public {
    vm.startPrank(ashley);
    TestWETH test_weth = new TestWETH();
    uint256 fee = 1;

    Snow newSnow = new Snow(address(test_weth), fee, collector);

    newSnow.buySnow(10);

    uint256 res = newSnow.balanceOf(ashley);
    assertEq(res, 100);
}
```

Also add the following contract to the import list. This contract is just for replicating a transfer call that fails and does not do any transfers.

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.24;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Snow} from "../../src/Snow.sol";

contract TestWETH is ERC20 {

    uint256 public counter;
    address public immutable owner;
    address public snow_addr;

    constructor() ERC20("MockWETH", "mWETH") {
        owner = msg.sender;
    }

    function transfer(address, uint256) public pure override returns (
       bool) {
        return true;
    }
}
```

Recommended Mitigation Add checks to ensure that the transfer function returns a true before continuing to execute the rest of the function

```
-   i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
   amount));
+   bool res = i_weth.safeTransferFrom(msg.sender, address(this), (
   s_buyFee * amount));
+   assert(res == true);
    _mint(msg.sender, amount);
```

[M-4] Malicious WETH address in `Snow` can allow user to buy token for free

Description

- The `Snow::buySnow()` calls external functions from `i_weth` contract. The `safeTransferFrom()` function is a function from OpenZeppelin's SafeErc20 interface and internally it calls the `transferFrom()` function of the ERC20 token contract.
- A malicious contract inheriting from ERC20 contract but overriding the `transferFrom()` function can contain attacker's own logic and return true even if no tranfer has been performed

```
@>  i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
   amount));
```

Risk **Likelihood**:

- Deployer of contract puts malicious contract address either deliberately or due to human errors

Impact:

- User can call `buySnow()` once with the amount needed for buying, but then the `safeTransferFrom()` function does nothing

Proof of Concept Add the following testcase to the `Snow.t.sol` test suite:

```
function test_buyForFree() public {
    vm.startPrank(ashley);
    TestWETH test_weth = new TestWETH();
    uint256 fee = 1;
    Snow newSnow = new Snow(address(test_weth), fee, collector);
    newSnow.buySnow(10);
    uint256 res = newSnow.balanceOf(ashley);
    assertEq(res, 100);
}
```

Also add the following contract to the import list. This is the malicious contract:

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.24;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Snow} from "../../src/Snow.sol";

contract TestWETH is ERC20 {

    uint256 public counter;
    address public immutable owner;
    address public snow_addr;

    constructor() ERC20("MockWETH", "mWETH") {
        owner = msg.sender;
    }

    function transferFrom(address, address, uint256) public override
        returns (bool) {
        return true;
    }
}
```

Recommended Mitigation Check for the balance of the contract before and after the transfer has been done, and add the check to ensure that transfer has been done successfully

```
+    uint256 before = i_weth.balanceOf(msg.sender);
-    i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
    amount));
+    bool res = i_weth.safeTransferFrom(msg.sender, address(this), (
    s_buyFee * amount));
+    assert(res == true);
+    uint256 after = i_weth.balanceOf(msg.sender);
+    assertEq(after, before - (s_buyFee * amount));
```

[M-4] Malicious WETH address in `Snow` can allow user to buy token for free

Description

- The `Snow::buySnow()` calls external functions from `i_weth` contract. The `safeTransferFrom()` function is a function from OpenZeppelin's SafeErc20 interface and internally it calls the `transferFrom()` function of the ERC20 token contract.
- A malicious contract inheriting from ERC20 contract but overriding the `transferFrom()` function can contain attacker's own logic and return true even if no tranfer has been performed

```
@>  i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
    amount));
```

Risk

Likelihood:

- Deployer of contract puts malicious contract address either deliberately or due to human errors

Impact:

- User can call `buySnow()` once with the amount needed for buying, but then the `safeTransferFrom()` function does nothing

Proof of Concept

Add the following testcase to the `Snow.t.sol` test suite:

```
function test_buyForFree() public {
    vm.startPrank(ashley);
    TestWETH test_weth = new TestWETH();
    uint256 fee = 1;
    Snow newSnow = new Snow(address(test_weth), fee, collector);
    newSnow.buySnow(10);
    uint256 res = newSnow.balanceOf(ashley);
    assertEq(res, 100);
}
```

Also add the following contract to the import list. This is the malicious contract:

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.24;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Snow} from "../../src/Snow.sol";

contract TestWETH is ERC20 {

    uint256 public counter;
    address public immutable owner;
    address public snow_addr;

    constructor() ERC20("MockWETH", "mWETH") {
        owner = msg.sender;
    }

    function transferFrom(address, address, uint256) public override
       returns (bool) {
        return true;
    }
}
```

Recommended Mitigation Check for the balance of the contract before and after the transfer has been done, and add the check to ensure that transfer has been done successfully

```
+    uint256 before = i_weth.balanceOf(msg.sender);
-    i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee *
    amount));
+    bool res = i_weth.safeTransferFrom(msg.sender, address(this), (
    s_buyFee * amount));
+    assert(res == true);
+    uint256 after = i_weth.balanceOf(msg.sender);
+    assertEq(after, before - (s_buyFee * amount));
```

[M-5] Inefficient loop usage in `mintSnowman()` can cause gas fees exceeding block gas limit

Description

- The `mintSnowman()` function uses a for loop to mint the NFTs
- Due to the unbound loop the execution is very gas costly and can exceed the block gas limit

```
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:

- Whenever a very high value for `amount` is passed while calling `mintSnowman()`

Impact:

- The execution may exceed block gas limit
- FUnction will revert
- The user who called it will not be able to mint any tokens, as reducing the amount of Snow tokens will cause their wallet to become ineligibe for airdrop

Proof of Concept Add the following test case to the test suite of snowman:

```
function test_gasInefficiency() public {
    uint256 start = gasleft();

    nft.mintSnowman(alice, 1);

    uint256 lessFee = start - gasleft();
    start = gasleft();

    nft.mintSnowman(alice, 1000);

    uint256 moreFee = start - gasleft();

    assert(lessFee < moreFee);
}
```

Recommended Mitigation

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

-       emit SnowmanMinted(receiver, s_TokenCounter);

-       s_TokenCounter++;
-   }

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

[M-6] `SnowmanAirdrop::_isValidSignature()` does not check for errors during `actualSigner` calculation

Description

- `_isValidFunction()` uses ECDSA interface to calculate signer from digest, v, r and s
- It does not take account the errors that can be generated during validation

```
function _isValidSignature(address receiver, bytes32 digest, uint8 v,
    bytes32 r, bytes32 s)
     internal
     pure
     returns (bool)
{
@>   (address actualSigner,,) = ECDSA.tryRecover(digest, v, r, s);
     return actualSigner == receiver;
}
```

Risk Likelihood:

- Whenever `tryRecover()` returns an error

Impact:

- Error not recognised and handled

Proof of Concept This is the function signature of `tryRecover()`:

```
function tryRecover(
     bytes32 hash,
     uint8 v,
     bytes32 r,
     bytes32 s
) internal pure returns (address recovered, RecoverError err, bytes32
    errArg) {
```

It returns three objects - `recovered`, `err` and `errArg`. `err` and `errArg` are used for error handling when errors are generated.

Recommended Mitigation

```
function _isValidSignature(address receiver, bytes32 digest, uint8 v,
    bytes32 r, bytes32 s)
     internal
     pure
     returns (bool)
{
-    (address actualSigner,,) = ECDSA.tryRecover(digest, v, r, s);
+    (address actualSigner, RecoverError err, bytes32 errArg) = ECDSA.
    tryRecover(digest, v, r, s);
+    if (err) {
+      revert();
+    }
```

```
    return actualSigner == receiver;
}
```

Low

[L-1] Non-specific version of solc can lead to unexpected compiler behaviour

Description

- Solidity updates and patch versions may introcude new changes in compiler behaviour, optimizations that may introduce bugs in the contract due to unexpected compiler behaviour

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
@> pragma solidity ^0.8.24;
```

Risk **Likelihood**:

- Whenever the contracts are compiled with newer versions of compiler

Impact:

- Unexpected behaviour

Recommended Mitigation

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
-pragma solidity ^0.8.24;
+pragma solidity 0.8.24;
```

[L-2] `Snow::earnSnow()` is not emitting the `SnowEarned()` event and event listeners will never know when a `Snow` is earned

Description

- The `earnSnow()` function allows user to earn `Snow` every week. However, no events are emitted when the function is called.

```
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:

- Whenever the `earnSnow()` event is called

Impact:

- Frontends depending on event emission to detect when a `Snow` is earned by a user will never be able to update because no events are emitted

Proof of Concept The `SnowEarned` event is defined in the contract like the following:

```
event SnowEarned(address indexed earner, uint256 indexed amount);
```

However, it has never been used

Recommended Mitigation: Modify the `earnSnow()` function to be like this

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

Now, every time a Snow is earned using this function, an event will be emitted

[L-3] `Snow::collectFee()` is not emitting the `FeeCollected()` event and event listeners will never know when fee is collected

Description

- The `collectFee()` function allows user to collect the fees accumulated. However, no events are emitted when the function is called.

```
function collectFee() external onlyCollector {
    uint256 collection = i_weth.balanceOf(address(this));
    i_weth.transfer(s_collector, collection);

    (bool collected,) = payable(s_collector).call{value: address(this).
        balance}("");
    require(collected, "Fee collection failed!!!");
}
```

Risk Likelihood:

- Whenever the `collectFee()` event is called to collect the fee from protocol

Impact:

- Frontends depending on event emission to detect when fee is collected will never be able to update because no events are emitted

Proof of Concept The `FeeCollected` event is defined in the contract like the following:

```
event FeeCollected();
```

However, it has never been used in the contract.

Recommended Mitigation: Modify the `collectFee()` function to be like this

```
function collectFee() external onlyCollector {
+   emit FeeCollected();
    uint256 collection = i_weth.balanceOf(address(this));
    i_weth.transfer(s_collector, collection);
    (bool collected,) = payable(s_collector).call{value: address(this).
       balance}("");
    require(collected, "Fee collection failed!!!");
}
```

Now, every time fee is collected using this function, an event will be emitted

[L-4] Not checking for success of transfer function in `Snow::collectFee()`

Description

- The `collectFee()` function is calling WETH token's transfer function to transfer all fees to the collector
- However, there are no checks to ensure that that the transfer is a success

```
function collectFee() external onlyCollector {
    uint256 collection = i_weth.balanceOf(address(this));
@>  i_weth.transfer(s_collector, collection);
    (bool collected,) = payable(s_collector).call{value: address(this).
       balance}("");
    require(collected, "Fee collection failed!!!");
}
```

Risk Likelihood:

- Whenever the `collectFee()` function is called and transfer fails

Impact:

- No fee transfer but function executed fully

Proof of Concept Add the following testcase to the test suite of `Snow`

```
function test_falseTransfer() public {
    TestWETH test_weth = new TestWETH();
    uint256 fee = 1;

    Snow newSnow = new Snow(address(test_weth), fee, collector);
    vm.startPrank(collector);
    newSnow.collectFee();
}
```

Also add the following contract to the import list. This contract is just to replicate a situation where the token transfer has failed.

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.24;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Snow} from "../../src/Snow.sol";

contract TestWETH is ERC20 {

    uint256 public counter;
    address public immutable owner;
    address public snow_addr;

    constructor() ERC20("MockWETH", "mWETH") {
        owner = msg.sender;
    }

    function transfer(address, uint256) public pure override returns (
        bool) {
        return false;
    }
}
```

Recommended Mitigation Add checks to ensure that the transfer function is a success

```
-    i_weth.transfer(s_collector, collection);
+    bool res = i_weth.transfer(s_collector, collection);
+   assert(res == true);
```

[L-5] No checks for amount value in `Snow::buySnow()` can allow for 0 amount purchases

Description

- In the `Snow::buySnow()` function, the `amount` variable determines the amount of Snow tokens to buy
- There are no checks for checking if the `amount` is a valid value

```
function buySnow(uint256 amount) external payable canFarmSnow {
    if (msg.value == (s_buyFee * amount)) {
        _mint(msg.sender, amount);
```

Risk

Likelihood:

- Whenever `buySnow()` is called with 0 amount

Impact:

- Invalid event emitted

Proof of Concept

Add the following test case to the test suite

```
function testCanBuySnowWithZeroAmount() public {
    vm.prank(victory);
    snow.buySnow(0);
}
```

Recommended Mitigation

Add this check at the beginning of `buySnow()` to revert if amount is 0

```
+   assert(amount != 0);
```

[L-6] `Snowman::mintSnowman` does not check for 0 amount

Description

- The `mintSnowman()` takes in parameter `amount` for determining the number of NFTs to mint
- The function does not check for 0 amount which basically does nothing

```
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**:

- Whenever `mintSnowman()` is called with `amount` = 0

Impact:

- Unnecessary function execution

Proof of Concept Add the following test case to the test suite of `Snowman`

```
function test_zeroAmount() public {
    nft.mintSnowman(alice, 0);
}
```

Recommended Mitigation

```
+ assert(amount != 0);
```

Informational

[INFO] Multiples of 1000 should be written with `e` notation

Description

- The value of `Snow::PRECISON` has been given as `10 ** 18`

```
@>  uint256 constant PRECISION = 10 ** 18;
    uint256 constant FARMING_DURATION = 12 weeks;
```

Recommended Mitigation

```
-   uint256 constant PRECISION = 10 ** 18;
+   uint256 constant PRECISION = 1e18;
    uint256 constant FARMING_DURATION = 12 weeks;
```

[INFO] Unused events

Description

- Events `Snow::SnowEarned()` and `Snow::FeeCollected()` are never used anywhere in the codebase

```
@>   event SnowEarned(address indexed earner, uint256 indexed amount);
@>   event FeeCollected();
```

Recommended Mitigation

```
-    event SnowEarned(address indexed earner, uint256 indexed amount);
-    event FeeCollected();
```

[INFO] `Snow::FeeCollected()` event should emit the amount of fee collected

Description

- Currently, the `FeeCollected()` event signature is defined like this:

```
     event FeeCollected();
```

Recommended Mitigation

```
-    event FeeCollected();
+    event FeeCollected(uint256 indexed amount);
```

[INFO] `Snowman::SM__NotAllowed()` error has been declared but never used in the codebase

Description

- The `SM__NotAllowed` error has been declared to revert and give custom error but it has never been used.

```
error SM__NotAllowed();
```

Recommended Mitigation

```
-    error SM__NotAllowed();
```

[INFO] Storage variable should be in mixedCase but some variables are not

Description

- `Snowman::s_TokenCounter` and `Snowman::s_SnowmanSvgUri` are not in mixedCase and do not follow the conventions followed in the rest of the codebase

```
uint256 private s_TokenCounter;
string private s_SnowmanSvgUri;
```

Recommended Mitigation

```
-   uint256 private s_TokenCounter;
+   uint256 private s_tokenCounter;
-   string private s_SnowmanSvgUri;
+   string private s_snowmanSvgUri;
```

[INFO] `SnowmanAirdrop::s_claimers` variable is not used anywhere in the codebase

Description

- `SnowmanAirdrop::s_claimers` variable has been declared but never used in the entire contract and can be deleted

```
address[] private s_claimers; // array to store addresses of claimers
```

Risk **Impact**:

- High gas fees during deployment
-

```
-   address[] private s_claimers; // array to store addresses of
    claimers
```

[INFO] Two `if` statement branches can be combined into one branch in `SnowmanAirdrop.sol`

Description

- There are two `if` statement branches in `SnowmanAirdrop::constructor()` returning the same error and performing similar checks which can be merged into one.

```
constructor(bytes32 _merkleRoot, address _snow, address _snowman)
    EIP712("Snowman Airdrop", "1") {
@>  if (_snow == address(0)) {
        revert SA__ZeroAddress();
    }
@>  if (_snowman == address(0)) {
        revert SA__ZeroAddress();
    }
```

Risk Likelihood:

- When constructor is called for deployment

Impact:

- More gas fees during deployment

Recommended Mitigation The two if statements can be merged into one like this:

```
constructor(bytes32 _merkleRoot, address _snow, address _snowman)
    EIP712("Snowman Airdrop", "1") {
-   if (_snow == address(0)) {
+   if (_snow == address(0) || _snowman == address(0)) {
        revert SA__ZeroAddress();
    }
-   if (_snowman == address(0)) {
-       revert SA__ZeroAddress();
-   }
```

[INFO] CEI pattern not being followed in `SnowmanAirdrop::claimSnowman()`

Description

- While transferring Snow tokens from user to contract then transferring Snowman NFT from contract to user, CEI pattern is not being followed

```
    i_snow.safeTransferFrom(receiver, address(this), amount); // send
        tokens to contract... akin to burning

    s_hasClaimedSnowman[receiver] = true;

    emit SnowmanClaimedSuccessfully(receiver, amount);

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

Recommended Mitigation

```
    s_hasClaimedSnowman[receiver] = true;

    emit SnowmanClaimedSuccessfully(receiver, amount);

    i_snow.safeTransferFrom(receiver, address(this), amount); // send
       tokens to contract... akin to burning

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

Gas

GAS Variable not changes after constructor calls can be made immutable to save gas fees

Description

- The `s_buyFee` and `i_weth` are being set in the constructor function and then never changed after that.
- These variables can be set to `immutable` to reduce gas fees

```
@>   uint256 public s_buyFee;
     uint256 private immutable i_farmingOver;

@>   IERC20 i_weth;
```

Risk **Likelihood**:

- Always

Impact:

- Slightly higher gas fees

Recommended Mitigation

```
-    uint256 public s_buyFee;
+    uint256 public immutable s_buyFee;
     uint256 private immutable i_farmingOver;

-    IERC20 i_weth;
+   IERC20 immutable i_weth;
```

GAS Variable not changed after constructor calls can be made immutable to save gas fees

Description

- The `s_SnowmanSvgUri` is being set in the constructor function and then never changed after that.
- These variables can be set to `immutable` to reduce gas fees

```
    string private s_SnowmanSvgUri;
```

Risk Likelihood:

- Always

Impact:

- Slightly higher gas fees

Recommended Mitigation

```
-   string private s_SnowmanSvgUri;
+   string private immutable s_SnowmanSvgUri;
```

GAS `Snowman::tokenUri()` is never called inside the contract even though it is a public function and consumes more gas

Description

- `Snowman::tokenUri()` is a public function which allows it to be called from within the contract too. But since it has never been used in the contract, it should be made `external` to save gas fees

```
    // >>> PUBLIC FUNCTIONS
@>    function tokenURI(uint256 tokenId) public view virtual override
   returns (string memory) {
        if (ownerOf(tokenId) == address(0)) {
            revert ERC721Metadata__URI_QueryFor_NonExistentToken();
        }
```

Risk Likelihood:

- Whenever `tokenUri()` is called

Impact:

- Slightly higher gas fees
-

```diff
     // >>> PUBLIC FUNCTIONS
-    function tokenURI(uint256 tokenId) public view virtual override
    returns (string memory) {
+    function tokenURI(uint256 tokenId) external view virtual override
    returns (string memory) {
         if (ownerOf(tokenId) == address(0)) {
             revert ERC721Metadata__URI_QueryFor_NonExistentToken();
         }
```