Protocol Audit Report

Azriel

June 18, 2025

Protocol Audit Report

Version 1.0

Cyfrin.io

June 18, 2025

Protocol Audit Report

Azriel

June 18, 2025

Prepared by: Azriel

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] `Snowman::mintSnowman()` Does Not Control Caller Allowing Anyone To Mint Arbitrary Number of NFTs To Themselves
    - Description
    - Risk
    - Proof of Concept
    - Recommended Mitigation
  - [H-2] `Snow::earnSnow()` Function Uses Shared `s_earnTimer` Allowing Only One Address To Earn Snow Per Week
    - Description
    - Risk
    - Proof of Concept
    - Recommended Mitigation
  - [H-3] `Snow::buySnow()` Updates Shared `s_earnTimer` Variable Causing Denial Of Service Within `Snow::earnSnow()`
    - Description
    - Risk
    - Proof of Concept
    - Recommended Mitigation
  - [H-4] `SnowmanAirdrop::claimSnowman()` Uses Current Token Balance For Merkle Proof Making It Susceptible to DoS

        * Description
        * Risk
        * Proof of Concept
        * Recommended Mitigation
- Low
    - [L-1] `Snow::buySnow()` Expects Exact Amount Of Ether, Resulting In Unexpected Reverts / WETH Transfer
        * Description
        * Risk
        * Recommended Mitigation
- Informational / Gas
    - I-1: Centralization Risk for trusted owners
    - I-2: Unsafe ERC20 Operations should not be used
    - I-3: Solidity pragma should be specific, not wide
    - I-4: `public` functions not used internally could be marked `external`
    - I-5: Event is missing `indexed` fields
    - I-6: PUSH0 is not supported by all chains
    - I-7: Unused Custom Error

Protocol Summary

This protocol is used to airdrop Snowman NFTs to Snow ERC20 token holders.

Disclaimer

The Azriel(me) 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

Scope

In Scope:

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

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

Roles

NA

Executive Summary

For this audit, I used about 7h in total. I made use of tools like Slither and Aderyn to first conduct static analysis, in addition to manual analysis. Additionally, I used cloc and Solidity Metrics to aid me in the initial scoping phase and understanding of the code base better. I had to research and understand merkle trees and their usecase within airdrops to conduct a more accurate audit on the airdrop portion of the protocol.

Issues found

Severity	Number of Issues Found
High	4
Medium	0
Low	1
Info	6

Findings

High

[H-1] `Snowman::mintSnowman()` Does Not Control Caller Allowing Anyone To Mint Arbitrary Number of NFTs To Themselves

Description

- Snowman NFTs should only be earned through the airdrop.
- `Snowman::mintSnowman()` can be exploited by anyone to mint Snowman NFTs to themselves as the function is not protected by any modifier

```
@> function mintSnowman(address receiver, uint256 amount) external { // not protected by mod
    for (uint256 i = 0; i < amount; i++) {
        _safeMint(receiver, s_TokenCounter);

        emit SnowmanMinted(receiver, s_TokenCounter);

        s_TokenCounter++;
    }
}
```

Risk

Likelihood: High

- It is very simple for anyone to call this function directly at any point of time without going throught the airdrop

Impact: High

- Since anyone can call this function with any arbitrary number of NFTs to be minted to themselves, the value of the NFT has basically become worthless

Proof of Concept

The following code proves that any address can mint any number of NFTs directly to themselves.

```
function testMintSnowmanDirectly() public {
    assert(nft.balanceOf(alice) == 0);
    vm.startPrank(alice);
    nft.mintSnowman(alice, 5);
    vm.stopPrank();
    assert(nft.balanceOf(alice) == 5);
```

```
}
```

Recommended Mitigation

The most flexible solution would be for the contract to inherit from OpenZeppelin's AccessControl to allow the owner of the contract to whitelist addresses that can call the `mintSnowman()` function.

```
- contract Snowman is ERC721, Ownable {
+ contract Snowman is ERC721, Ownable, AccessControl {

+ bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

constructor() ERC721("Snowman", "SNOW") Ownable(msg.sender) {
    // original code...
+    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender); // Admin can grant MINTER_ROLE to airdrop c
}

- function mintSnowman(address receiver, uint256 amount) external {
+ function mintSnowman(address receiver, uint256 amount) external onlyRole(MINTER_ROLE) {


// Called after airdrop contract is deployed
+ function setAirdropContract(address airdropContract) external onlyRole(DEFAULT_ADMIN_ROLE)
+     _grantRole(MINTER_ROLE, airdropContract);
+ }

// Remove minting ability (if required)
+ function renounceAdminMinting() external onlyRole(DEFAULT_ADMIN_ROLE) {
+     _revokeRole(MINTER_ROLE, msg.sender);
+ }
```

[H-2] `Snow::earnSnow()` Function Uses Shared `s_earnTimer` Allowing Only One Address To Earn Snow Per Week

Description

- All addresses should be able to call the `Snow::earnSnow()` function during the farming phase once a week to earn Snow tokens before the airdrop.
- When the first address calls the function, the `Snow::s_earnTimer` variable is updated for the whole contract, thus disallowing any other address from calling this function to earn Snow tokens.

```
    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: High

- This is a common function that everyone has access to and would use before the airdrop.

Impact: High

- This would effectively cause a Denial-of-Service as others who legitimately want to earn Snow tokens would cause a revert.

Proof of Concept

The following test was added to `TestSnow.sol` to demonstrate the issue.

```
function testOnlyOneCanEarnSnow() public {
    vm.prank(ashley);
    snow.earnSnow();

    assert(snow.balanceOf(ashley) == 1);

    vm.prank(jerry);
    vm.expectRevert(); // Should not revert as Jerry has not earned his free Snow token yet,
    snow.earnSnow();
}
```

Recommended Mitigation

Instead of all addresses sharing a single `s_earnTimer` variable, a mapping can be used for each address to store the last claimed time for each address.

```
- uint256 private s_earnTimer;
+ mapping(address => uint256) private earnTimer
```

Under `earnSnow()`:

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

-    s_earnTimer = block.timestamp;
```

```
+     earnTimer[msg.sender] = block.timestamp;
}
```

[H-3] `Snow::buySnow()` Updates Shared `s_earnTimer` Variable Causing Denial Of Service Within `Snow::earnSnow()`

Description

- `Snow::buySnow()` should allow Snow tokens to be bought at anytime, while `Snow::earnSnow()` should allow users to earn Snow tokens for free once a week.
- The issue here is that `Snow::buySnow()` updates the `s_earnTimer` variable which is also used within `Snow::earnSnow()` to check if a week has passed before allowing users to earn Snow tokens. This results in a Denial Of Service for users to call `Snow::earnSnow()`

```
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; // variable updated here

    emit SnowBought(msg.sender, amount);
}

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

    s_earnTimer = block.timestamp;
}
```

Risk

Likelihood: High

- `Snow::buySnow()` is an advertised component which would be used often by users

Impact: High

- Users will not be able to earn Snow tokens which is an advertised component of the airdrop.

Proof of Concept

The following test added to `TestSnow.t.sol` demonstrates the Denial Of Service for the `earnSnow()` function when `buySnow()` is called by anyone.

```
function testCannotEarnSnowAfterBuySnow() public {
    vm.prank(victory);
    snow.buySnow{value: FEE}(1);

    assert(victory.balance == 0);
    assert(address(snow).balance == FEE);
    assert(snow.balanceOf(victory) == 1);

    vm.prank(ashley);
    vm.expectRevert(); // Should not revert as Ashley has not earned free Snow token yet, bu
    snow.earnSnow();

    vm.warp(block.timestamp + 1 weeks);

    vm.prank(ashley);
    snow.earnSnow();
    assert(snow.balanceOf(ashley) == 1);
}
```

Recommended Mitigation

This issue can be mitigated by removing the line that updates `s_earnTimer` within `Snow::buySnow()`.

```
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);
}
```

[H-4] `SnowmanAirdrop::claimSnowman()` Uses Current Token Balance For Merkle Proof Making It Susceptible to DoS

Description

- In the normal case, users would earn Snow tokens through the Snow contract and would not have their balances changed after the farming phase. The current `amount` of Snow tokens a user would have when calling `claimSnowman()` would be the same as the `amount` that was used to generate the merkle tree.
- However this would also make the function susceptible to DoS attacks as bad actors can manually transfer tokens after the farming phase to cause a mismatch between current `amount` and the `amount` used to generate the merkle tree, causing merkle proof to fail.

```
function claimSnowman(address receiver, bytes32[] calldata merkleProof, uint8 v, bytes32 r,
    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); // uses current token balance

@>    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...

    s_hasClaimedSnowman[receiver] = true;

    emit SnowmanClaimedSuccessfully(receiver, amount);

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

Risk

Likelihood: High

- It would be very simple for attackers to transfer ERC20 tokens manually if they want to cause a DoS to a particular address.

Impact: Medium

- Users may not understand why their claims are failing, but it could also be solved by manually transferring excess tokens out of their wallets so that their balance aligns with the original balance.

Proof of Concept

The following test case shows how alice is no longer able to claim once bob transfers additional Snow tokens to her.

```
function testDenialOfService() public {
    // Alice claim test
    assert(nft.balanceOf(alice) == 0);
    vm.prank(alice);
    snow.approve(address(airdrop), 1);

    // Get alice's digest
    bytes32 alDigest = airdrop.getMessageHash(alice);

    // alice signs a message
    (uint8 alV, bytes32 alR, bytes32 alS) = vm.sign(alKey, alDigest);

    // attacker sends an additional Snow token to Alice so that the merkle proof fails
    vm.prank(bob);
    snow.transfer(alice, 1);

    assert(snow.balanceOf(alice) == 2);
    assert(snow.balanceOf(bob) == 0);

    // satoshi tries to claim on behalf of alice using her signed message
    vm.prank(satoshi);
    vm.expectRevert();
    airdrop.claimSnowman(alice, AL_PROOF, alV, alR, alS);

    assert(nft.balanceOf(alice) == 0);
}
```

Recommended Mitigation

The function should accept an additional variable for users to input how much tokens they own, which should correspond to the amount used to build the

merkle tree with. The drawback of this is that users would need to track how many tokens they own at the end of the farming phase and before the airdrop.

```
- function claimSnowman(address receiver, bytes32[] calldata merkleProof, uint8 v, bytes32 r
+ function claimSnowman(address receiver, uint256 claimAmount, bytes32[] calldata merkleProo
    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))));
+    bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(receiver, claimAmount))));

    if (!MerkleProof.verify(merkleProof, i_merkleRoot, leaf)) {
        revert SA__InvalidProof();
    }
-    i_snow.safeTransferFrom(receiver, address(this), amount); // send tokens to contract...
+    i_snow.safeTransferFrom(receiver, address(this), claimAmount);
    s_hasClaimedSnowman[receiver] = true;

-    emit SnowmanClaimedSuccessfully(receiver, amount);
+    emit SnowmanClaimedSuccessfully(receiver, claimAmount);

-    i_snowman.mintSnowman(receiver, amount);
+    i_snowman.mintSnowman(receiver, claimAmount);
}
```

Low

[L-1] `Snow::buySnow()` Expects Exact Amount Of Ether, Resulting In Unexpected Reverts / WETH Transfer

Description

- This protocol should allow users to clearly choose between paying with ether or weth.
- The same function however, is being used. The function determines whether a user chooses ether or weth by checking if the `msg.value` is exactly equal to the amount required. If users estimate the amount wrongly, their transaction would revert or would accidentally trigger a weth transfer if the user has approved the protocol from spending their weth previously.

```
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: Low

- Assuming that the `s_buyFee` is a whole number, users should not have difficulty calculating the price exactly.

Impact: Low

- Most likely the protocol would revert, unless user has approved the protocol from spending weth previously. Thus there would be minimal impact on the user, besides usability issues.

Recommended Mitigation

A good practice would be to have separate functions, and return excess ether to the user if `msg.value` > amount required.

```
- 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);
- }
+ function buySnowEth(uint256 amount) external payable canFarmSnow {
+    require (msg.value >= (s_buyFee * amount), "Insufficient ether sent!");
+    uint256 balance = msg.value - s_buyFee * amount;
+    if (balance > 0) {
+       (bool return,) = payable(msg.sender).call{value: balance}("");
+       require(return, "Fee return failed!!!");
+    }
+    _mint(msg.sender, amount);
+
+    s_earnTimer = block.timestamp;
+
+    emit SnowBought(msg.sender, amount);
+ }

+ function buySnowWeth(uint256 amount) external canFarmSnow {
+    i_weth.safeTransferFrom(msg.sender, address(this), (s_buyFee * amount));
+    _mint(msg.sender, amount);
+
+    s_earnTimer = block.timestamp;
+
+    emit SnowBought(msg.sender, amount);
+ }
```

Informational / Gas

I-1: Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

2 Found Instances

- Found in src/Snow.sol Line: 18

  ```
  contract Snow is ERC20, Ownable {
  ```

- Found in src/Snowman.sol Line: 17

  ```
  contract Snowman is ERC721, Ownable {
  ```

I-2: Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

1 Found Instances

- Found in src/Snow.sol Line: 109

```
i_weth.transfer(s_collector, collection);
```

I-3: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

4 Found Instances

- Found in src/Snow.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/Snowman.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/SnowmanAirdrop.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/mock/MockWETH.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

I-4: `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

1 Found Instances

- Found in src/Snowman.sol Line: 48

```
function tokenURI(uint256 tokenId) public view virtual override returns (string mem
```

I-5: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and

gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

1 Found Instances

- Found in src/SnowmanAirdrop.sol Line: 52

```
event SnowmanClaimedSuccessfully(address receiver, uint256 amount); // emitted when
```

I-6: PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

4 Found Instances

- Found in src/Snow.sol Line: 2

```
pragma solidity ^0.8.24;
```

- Found in src/Snowman.sol Line: 2

```
pragma solidity ^0.8.24;
```

- Found in src/SnowmanAirdrop.sol Line: 2

```
pragma solidity ^0.8.24;
```

- Found in src/mock/MockWETH.sol Line: 2

```
pragma solidity ^0.8.24;
```

I-7: Unused Custom Error

it is recommended that the definition be removed when custom error is unused

1 Found Instances

- Found in src/Snowman.sol Line: 20

```
error SM__NotAllowed();
```