Protocol Audit Report

Version 1.0

XV Dev Labs

April 7, 2025

Eggstravaganza Audit Report

XV Dev Labs

April 7, 2025

Prepared by: XV Dev Labs

Lead Auditors: - ArefXV

Table of Contents

  - [L-1] Unspecific Solidity Pragma
  - [L-2] Public Function Not Used Internally
  - [L-3] State Change Without Event

- Informational
  - [I-1] Magic Numbers
  - [I-2] State Variable Could Be Immutable
  - [I-3] Test Coverage
  - [I-4] String Return Type Increases Gas Consumption

Protocol Summary

EggHuntGame is a gamified NFT experience where participants search for hidden eggs to mint unique Eggstravaganza Egg NFTs. Players engage in an interactive hunt during a designated game period, and successful egg finds can be deposited into a secure Egg Vault.

Disclaimer

The XV Dev Labs 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
Likelihood	High	H	H/M	M
	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.

Scope

```
src/
#-- EggHuntGame.sol       // Main game contract managing the egg hunt lifecycle and minting process.
#-- EggVault.sol          // Vault contract for securely storing deposited Egg NFTs.
#-- EggstravaganzaNFT.sol // ERC721-style NFT contract for minting unique Egg NFTs.
```

Roles

Actors: Game Owner: The deployer/administrator who starts and ends the game, adjusts game parameters, and manages ownership. Player: Participants who call the egg search function, mint Egg NFTs upon successful searches, and may deposit them into the vault. Vault Owner: The owner of the EggVault contract responsible for managing deposited eggs.

Issues found

Severity	Issues Found
High	1
Medium	2
Low	3
Informational	4
Total	10

Findings

High

[H-1] Weak Randomness in `EggHuntGame::searchForEgg` function migh lead to game manipulation risk

Description: The contract relies on `block.timestamp`, `block.prevrandao`, and `msg.sender` to generate pseudo-random numbers. These values can be partially controlled or predicted by mali-

cious actors (e.g., miners), especially in a public network, making it possible to manipulate the result of `EggHuntGame::searchForEgg` function.

Impact: A malicious player or miner could increase their chances of finding an egg, gaining unfair advantage in the game, and disrupting game fairness and token economy

Proof of Concept:

```
uint256 random = uint256(
    keccak256(abi.encodePacked(block.timestamp, block.prevrandao, msg.
        sender, eggCounter))
) % 100;

if (random < eggFindThreshold) {
    eggCounter++;
    eggsFound[msg.sender] += 1;
    eggNFT.mintEgg(msg.sender, eggCounter);  // unfair minting
        opportunity
}
```

An attacker could call `EggHuntGame::searchForEgg` function in a block they mine themselves or repeatedly call it across multiple transactions and blocks to eventually hit a winning result

Recommended Mitigation: Avoid relying on block variables for randomness. Instead:

- Use Chainlink VRF for secure randomness.
- Or, if full trustless randomness isn't required, make randomness off-chain and verifiable (signed results from backend + ECDSA).

Medium

[M-1] Event Emitted After External Call May Obscure Reentrancy or Failure

Description: The `EggVault::withdrawEgg` function emits an event **after** making an external call, which could lead to off-chain desynchronization. If the call fails but the event still gets emitted (e.g., due to low-level call not reverting properly or fallback behaviors), indexers like The Graph or any backend that listens to events might assume the transaction succeeded, while it didn't.

Impact: Incorrect off-chain state or user balance tracking, UI bugs, or even false confirmations to users.

Proof of Concept:

```
(bool success, ) = msg.sender.call{value: eggValue}("");
require(success, "Transfer failed");
```

```
3
4 // event emitted after call
5 emit EggWithdrawn(msg.sender, eggValue);
```

Recommended Mitigation:

1. emit the event before external call:

```
1  function withdrawEgg(uint256 tokenId) public {
2          require(storedEggs[tokenId], "Egg not in vault");
3          require(eggDepositors[tokenId] == msg.sender, "Not the original
               depositor");
4
5          storedEggs[tokenId] = false;
6          delete eggDepositors[tokenId];
7
8  +       emit EggWithdrawn(msg.sender, tokenId);
9          eggNFT.transferFrom(address(this), msg.sender, tokenId);
10 -        emit EggWithdrawn(msg.sender, tokenId);
11     }
```

2. use `nonReentrant` modifier from `ReentrancyGuard.sol` from “Openzeppelin”.

[M-2] Use of `_mint()` instead of `_safeMint()` can lead to tokens being locked in contracts

Description: Using `ERC721::_mint()` allows minting tokens to any address, including contracts that do not implement the `IERC721Receiver` interface. If a token is minted to a smart contract that does not properly handle `ERC721` tokens, the token may become permanently inaccessible or locked. The `_safeMint()` function performs additional checks to ensure the recipient is capable of receiving `ERC721` tokens.

Impact: Tokens might get locked in contracts that are not designed to handle ERC721 tokens, resulting in permanent loss of assets. This could also cause unexpected behavior and poor user experience.

Proof of Concept:

```
1 _mint(address(0xdead), tokenId); // 0xdead might be a contract not
      implementing ERC721Receiver
```

This call will succeed even if the recipient is not a valid ERC721 receiver contract!

Recommended Mitigation: Use `_safeMint()` instead of `_mint()` in `EggstravaganzaNFT::mintEgg` function when minting ERC721 tokens to ensure the receiving contract is able to handle the tokens correctly:

```
1 _safeMint(to, tokenId);
```

This ensures that if msg.sender is a contract, it must implement the onERC721Received interface, otherwise the mint will revert.

Low

[L-1] Unspecific Solidity Pragma

Impact: This issue can lead to unexpected behavior if the compiler version used changes in the future. While the code may work as expected with the specified version range (e.g., `^0.8.23`), the pragma can allow the contract to compile with newer versions that might introduce breaking changes or unexpected behavior. This can result in subtle bugs or vulnerabilities that are difficult to detect. Furthermore, using an imprecise version can make it harder to verify the contract's security and functionality with specific compiler version

Recommended Mitigation: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of pragma solidity `^0.8.23;`, use pragma solidity `0.8.23;`

[L-2] Public Function Not Used Internally

Description: If a function is marked `public` but is not called internally by other functions in the contract, it can be more gas-efficient and semantically clear to mark it as external. This is because `external` functions use less gas when called externally and do not need to be stored in the contract's internal function selector.

Impact: Marking a function as `public` when it's not used internally can increase gas costs unnecessarily, as `external` function calls are more optimized in terms of gas usage. Furthermore, using external where appropriate improves the readability of the contract by making it clear that the function is intended for external calls only.

Proof of Concept: For Instances, a function like `EggVault::depositEgg` can be marked as `external` instead of `public` if it is not called internally in the contract. Changing public to `external` reduces gas costs and clarifies the intended use of the function.

Recommended Mitigation: Change the visibility of the functions from `public` to `external` if they are only intended to be called externally (not inside the contract itself).

[L-3] State Change Without Event

Description: There are state variable changes in some functions like `EggHuntGame::setEggFindThreshold` (i.e., updating the `eggFindThreshold`) but no event is emitted to reflect these changes. Emitting events when state changes occur is essential for tracking contract interactions off-chain. Events allow external services and dApps to listen for specific changes in contract state, providing better transparency and traceability.

Impact: Lack of event emission makes it harder to track the contract's state changes, which may lead to difficulties in building frontend applications. It also reduces the ability for external indexers or monitoring systems to track important contract activities.

Recommended Mitigation: Emit events whenever state changes happen. In this case, an event can be added to notify when new egg find threshold will be set:

```
+ event NewThresholdSet(uint256 newThreshold);

function setEggFindThreshold(uint256 newThreshold) external onlyOwner {
        require(newThreshold <= 100, "Threshold must be <= 100");
        eggFindThreshold = newThreshold;
+       emit NewThresholdSet(uint256 newThreshold);
    }
```

Informational

[I-1] Magic Numbers

Description: All literal numbers should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "Magic Numbers".

Recommended Mitigation: For example:

```
+   uint256 public constant MAX_THRESHOLD = 100;

-   require(newThreshold <= 100, "Threshold must be <= 100");
```

Replace `MAX_THRESHOLD` with `100`.

[I-2] State Variable Could Be Immutable

Recommended Mitigation: State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

```
    EggstravaganzaNFT public immutable i_eggNFT;
```

[I-3] Test Coverage

Description: Branches coverage of the test of `EggHuntGame.sol` is below 90%. This often means that there are parts of this contract that are not tested.

Proof of Concept:

File | % Lines | % Statements | % Branches | % Funcs |

+===

| src/EggHuntGame.sol | 97.67% (42/43) | 97.37% (37/38) | 70.37% (19/27) | 100.00% (8/8) | |————————+————————+————————+————————+————————| | src/EggVault.sol | 100.00% (18/18) | 100.00% (14/14) | 90.00% (9/10) | 100.00% (4/4) | |————————————+————————+————————+————————+————————| | src/EggstravaganzaNFT.sol | 100.00% (8/8) | 100.00% (6/6) | 100.00% (4/4) | 100.00% (2/2) | |————————————+————————+————————+————————+————————| | Total | 98.55% (68/69) | 98.28% (57/58) | 78.05% (32/41) | 100.00% (14/14) |

Recommended Mitigation: Increase test coverage to 90% or higher.

[I-4] String Return Type Increases Gas Consumption

Description: The `EggHuntGame::getGameStatus` function returns a human-readable string, which is not gas-efficient. Using strings in smart contracts, especially as return types, can significantly increase gas costs and should be avoided unless absolutely necessary.

Recommended Mitigation: Return a `uint8` or a custom `enum` instead of a string, which can then be interpreted off-chain.

```
+   enum GameState { NotStarted, Active, Elapsed, Inactive }

function getGameStatus() external view returns (GameState) {
        if (gameActive) {
            if (block.timestamp < startTime) {
-               return "Game not started yet";
+               return GameState.NotStarted;
            } else if (block.timestamp >= startTime && block.timestamp
                <= endTime) {
-               return GameState.Active;
+               return true;
            } else {
-               return "Game time elapsed";
+               return GameState.Elapsed;
            }
        } else {
-            return "Game is not active";
+            return GameState.Inactive;
        }
    }
```