Aje’s Audit Report

Version 1.0

successaje.github.io

April 8, 2025

Aje's Audit Report

successaje.github.io

April, 2025

Prepared by: Aje Lead Auditors: - Success Aje

Table of Contents

- Gas
  - [G-1] Unchanged State Variable Should Be Declared as Constants or Immutable
  - [G-2] Unused Functions (EggVault)
  - [G-3] Literal Instead of Constant
- Informational
  - [I-1] Missing Critical Events
  - [I-2] Public Function `EggVault::depositEgg(uint256 tokenId, address depositor)` Not Used Internally
  - [I-3] Lack of Access Control in `EggVault::depositEgg(uint256 tokenId, address depositor)`
  - [I-4] Potential Reentrancy, State Change After External Call in `EggVault::depositEgg`

Protocol Summary

The Eggstravaganza protocol is a blockchain-based game designed to engage users in an interactive egg hunt. The protocol consists of three main smart contracts:

1. **EggHuntGame.sol**:
   This contract manages the lifecycle of the egg hunt, including player interactions, egg discovery, and game logic. It ensures fair gameplay by coordinating the rules and outcomes of the egg hunt.
2. **EggVault.sol**:
   The vault contract securely stores deposited Egg NFTs. It acts as a repository for eggs collected during the game, ensuring that they are safely held until they are withdrawn or utilized.
3. **EggstravaganzaNFT.sol**:
   This contract implements an ERC721-compliant NFT standard for minting unique Egg NFTs. Each egg represents a collectible item in the game, with unique identifiers and metadata.

The protocol is designed to be compatible with Ethereum and other EVM-compatible blockchains, leveraging the security and transparency of decentralized networks. By combining gamification with blockchain technology, Eggstravaganza aims to create an engaging and secure experience for its users.

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 me 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

```
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.
```

Compatibilties

```
Compatibilities:
  Blockchains:
      - Ethereum / Any EVM-compatible chain
  Tokens:
      - Custom ERC721 EggstravaganzaEggNFT tokens
```

Executive Summary

Executive Summary

The audit of the Eggstravaganza project identified several critical, high, and informational issues across its smart contracts. The findings primarily focus on security vulnerabilities, gas optimizations, and code quality improvements. Below is a summary of the key observations:

Key Findings:

1. **High Severity Issues**:
    - **Reentrancy Risk in `EggVault::depositEgg()`**: State changes occur after external calls, exposing the contract to potential reentrancy attacks.
    - **Unrestricted NFT Minting in `EggstravaganzaNFT::mintEgg()`**: Lack of validation for token uniqueness and recipient address could lead to duplicate tokens or lost NFTs.
    - **Weak Randomness in `EggHuntGame::searchForEgg()`**: Predictable randomness mechanism undermines the fairness of the game.
    - **Unchecked Return in `EggHuntGame::searchForEgg()`**: Failure to verify the success of critical operations could lead to logical inconsistencies.
2. **Low Severity Issues**:

Issues found

```
- **Unspecific Solidity Pragma**: Using a broad version range increases
  the risk of compatibility issues.
- **Unused Functions in `EggVault`**: Functions like `withdrawEgg()`
  and `isEggDeposited()` contribute to unnecessary code bloat.
```

3. **Gas Optimizations**:
    - **Immutable Variables**: Declaring frequently accessed state variables as `immutable` can reduce gas costs.
    - **Literal Constants**: Replacing repeated literals with constants improves readability and reduces deployment costs.

4. **Informational Observations**:
    - **Missing Events**: Critical administrative functions lack event emissions, reducing transparency.
    - **Access Control**: Functions like `EggVault::depositEgg()` lack proper access restrictions, increasing the attack surface.
    - **Potential Reentrancy Patterns**: External calls followed by state changes require additional safeguards.

Recommendations:

- Implement secure coding practices, such as reordering state changes and external calls, to mitigate reentrancy risks.
- Use Chainlink VRF or similar solutions for secure randomness.
- Add proper access control and event emissions to enhance transparency and security.
- Optimize gas usage by leveraging `immutable` variables and constants.
- Remove unused functions to reduce code bloat and improve maintainability.

The audit highlights the importance of addressing these issues to ensure the security, efficiency, and reliability of the Eggstravaganza smart contracts.

Severity	Number of issues found
High	4
Medium	0
Low	1
Info	7
Total	12

Findings

High

[H-1] Reentrancy Risk in Deposit (EggVault)

Description:
The `depositEgg()` function in the `EggVault` contract performs state changes after an external

call to the ownerOf function of the eggNFT contract. This introduces a reentrancy risk, as malicious contracts could exploit this sequence to manipulate the state of the vault.

Impact:
High - Reentrancy attacks could lead to unauthorized state changes, such as incorrect egg deposits or withdrawals, compromising the integrity of the vault.

Proof of Concept:

```
function depositEgg(uint256 tokenId) external {
    require(eggNFT.ownerOf(tokenId) == msg.sender, "Not the owner"); //
        External call
    storedEggs[tokenId] = true; // State change after external call
}
```

Recommended Mitigation:
Reorder the operations in the depositEgg() function to perform state changes before making external calls. For example:

```
function depositEgg(uint256 tokenId, address depositor) public {
    storedEggs[tokenId] = true; // State first
    eggDepositors[tokenId] = depositor;
    require(eggNFT.ownerOf(tokenId) == address(this)); // Check after
    emit EggDeposited(depositor, tokenId);
}
```

[H-2] Unrestricted NFT Minting (EggstravaganzaNFT)

Description:
The mintEgg() function in the EggstravaganzaNFT contract does not validate the uniqueness of the tokenId being minted or ensure that the recipient address is valid. This could lead to duplicate token IDs being minted or tokens being sent to the zero address, which would result in lost NFTs.

Impact:
High - Duplicate token IDs could compromise the integrity of the NFT collection, and minting to the zero address would result in irretrievable NFTs.

Proof of Concept:

```
function mintEgg(address to, uint256 tokenId) external {
    // No checks for tokenId uniqueness or recipient validity
    _mint(to, tokenId);
}
```

Recommended Mitigation:
Update the mintEgg() function to include the following checks: 1. Ensure the recipient address is

not the zero address. 2. Validate that the `tokenId` does not already exist.

```
function mintEgg(address to, uint256 tokenId) external {
    require(to != address(0), "Invalid recipient");
    require(!_exists(tokenId), "Token already exists");
    _safeMint(to, tokenId); // Use _safeMint for added safety
    totalSupply += 1;
    emit EggMinted(to, tokenId); // Emit an event for transparency
}
```

[H-3] Weak Randomness in `EggHuntGame::seachForEgg()`

Description:
The `searchForEgg()` function in the `EggHuntGame` contract uses `msg.sender`, `block.timestamp`, `block.prevrandao` and `eggCounter` to generate randomness for determining the outcome of an egg hunt. These values are predictable and can be manipulated by miners, leading to potential exploitation.

Impact:
High - Weak randomness can be exploited by malicious actors to predict or manipulate the outcome of the egg hunt, compromising the fairness and integrity of the game.

Proof of Concept:
Validators can know ahead of time the outcome of the `searchForEgg()` function by manipulating the block timestamp or prevrandao value. This predictability allows them to exploit the randomness mechanism to their advantage, undermining the fairness of the game.

```
function searchForEgg(uint256 playerId) external returns (bool) {
    // ......code.........
    uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp
        , block.prevrandao, , msg.sender, eggCounter))) % 100; // Weak
        Randomness
    // ......code.........
}
```

Recommended Mitigation:
Replace the weak randomness with a secure source of randomness, such as Chainlink VRF. For example:

```
// Example using Chainlink VRF
function searchForEgg(uint256 playerId) external returns (bool) {
    uint256 random = getSecureRandomNumber(playerId);
    return random % 2 == 0;
}

```

```
function getSecureRandomNumber(uint256 playerId) internal returns (
    uint256) {
    // Implement Chainlink VRF integration here
}
```

[H-4] Unchecked Return in `EggHuntGame::searchForEgg()`

Description:
The `searchForEgg()` function in the `EggHuntGame` contract calls `eggNFT.mintEgg()` without checking its return value. This could lead to logical inconsistencies if the minting process fails silently.

Impact:
1. If `mintEgg()` fails but does not revert, the contract will assume the minting was successful, leading to incorrect state updates. 2. The `EggFound` event will be emitted even if minting fails, creating false positives and potential abuse scenarios. 3. Silent failures could result in undetected issues, compromising the integrity of the contract.

Proof of Concept:

```
function searchForEgg() external {
    // ...code...
    if (random < eggFindThreshold) {
        eggCounter++;
        eggsFound[msg.sender] += 1;

        eggNFT.mintEgg(msg.sender, eggCounter); // Return value not
            checked

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

Recommended Mitigation:
Check the return value of `mintEgg()` and ensure it succeeds before proceeding with state updates or emitting events. For example:

```
function searchForEgg() external {
    // ...code...
    if (random < eggFindThreshold) {
        eggCounter++;
        eggsFound[msg.sender] += 1;

        bool success = eggNFT.mintEgg(msg.sender, eggCounter);
        require(success, "Egg minting failed");

```

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

This ensures that the contract handles minting failures gracefully and maintains a consistent state.

Low

[L-1] Unspecific Solidity Pragma

Contracts Affected:

- `EggstravaganzaNFT.sol`
- `EggVault.sol`
- `EggHuntGame.sol`

Description:
The contracts use an unspecific Solidity pragma (`pragma solidity ^0.8.23;`), which allows the code to compile with any version of Solidity from 0.8.23 onwards. This can lead to unexpected behavior if the code is compiled with a newer version of Solidity that introduces breaking changes or different behavior.

Impact:
Using an unspecific pragma increases the risk of compatibility issues and unexpected behavior when the code is compiled with a newer version of Solidity.

Proof of Concept:

```
pragma solidity ^0.8.23; // Unspecific pragma
```

Recommended Mitigation:
Specify a fixed Solidity version to ensure consistent behavior and compatibility. For example:

```
pragma solidity 0.8.23; // Specific pragma
```

This ensures that the code is compiled with the intended version of Solidity, reducing the risk of unexpected behavior.

Gas

[G-1] Unchanged state variable should be declared as constants or immutable in `EggHuntGame.sol`

Instances: - `eggNFT` should be `immutable` - `eggVault` should be `immutable`

The variable `eggNFT` is used to reference the NFT contract, and `eggVault` is used to reference the vault contract. These variables are not expected to change after deployment. Declaring them as `immutable` ensures that their values are set only once during contract initialization and cannot be modified afterward. This reduces gas costs associated with accessing these variables, as reading from `immutable` variables is cheaper than reading from storage.

Impact: Declaring state variables as `immutable` reduces gas costs by avoiding storage reads. This optimization is particularly beneficial for frequently accessed variables, as it improves contract efficiency and reduces transaction costs.

```
// Example of declaring variables as immutable
contract EggVault {
    address public immutable eggNFT;
    address public immutable eggVault;

    constructor(address _eggNFT, address _eggVault) {
        eggNFT = _eggNFT;
        eggVault = _eggVault;
    }
}
```

Recommended Mitigation: To address the issue, update the contract as follows:

1. Declare the variables as `immutable` in the contract.
2. Modify the constructor to initialize these variables.

```
// Updated Example
contract EggVault {
    address public immutable eggNFT;
    address public immutable eggVault;

    constructor(address _eggNFT, address _eggVault) {
        require(_eggNFT != address(0), "Invalid NFT address");
        require(_eggVault != address(0), "Invalid Vault address");
        eggNFT = _eggNFT;
        eggVault = _eggVault;
    }
}
```

This ensures that the variables are set only once during deployment and cannot be modified later, reducing gas costs and improving contract efficiency.

[G-2] Unused Functions (EggVault)

Description:
The functions `withdrawEgg()` and `isEggDeposited()` are defined in the contract but are not utilized in the current implementation. This results in unnecessary code bloat, which can increase deployment costs and reduce code readability.

Impact:
While there is no direct security impact, unused functions contribute to increased gas costs during deployment and make the codebase harder to maintain.

Proof of Concept:

```
function withdrawEgg(uint256 tokenId) external {
    // Unused function logic
}

function isEggDeposited(uint256 tokenId) external view returns (bool) {
    // Unused function logic
}
```

Recommended Mitigation:
1. Remove the unused functions if they are not required for the current or future implementation.
2. Alternatively, document these functions as utility functions if they are intended for future use or testing purposes.

[G-3] Literal instead of constant

Instances: `100` should be a constant in `EggHuntGame::setEggFindThreshold()`. `100` should be a constant in `EggHuntGame::searchForEgg()`.

Description: Define and use `constant` variables instead of using literals like `100`. If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract as it is cheaper to use that way

Impact: Gas - Code bloat but no direct security impact.

Proof of Concept:

```
uint256 constant MAX_EGGS = 100;
```

Recommended Mitigation: Replace all occurrences of the literal `100` with a defined constant variable such as `MAX_EGGS`.

Informational

[I-1] Missing Critical Events

Description:
Several critical administrative functions in the contracts do not emit events when executed. This lack of transparency makes it difficult to track and audit changes to important contract parameters, which could lead to reduced trust and accountability.

Impact:
Informational - Missing events for critical actions reduce transparency and make it harder to monitor and audit administrative changes.

Proof of Concept:
The following functions are missing event emissions: - `EggstravaganzaNFT.setGameContract(address newContract)` - `EggVault.setEggNFT(address newNFT)` - `EggHuntGame.setEggFindThreshold(uint256 newValue)`

Recommended Mitigation:
Add event declarations and emit the corresponding events in the affected functions. For example:

```
// EggstravaganzaNFT
event GameContractUpdated(address indexed newContract);

function setGameContract(address newContract) external onlyOwner {
    require(newContract != address(0), "Invalid address");
    gameContract = newContract;
    emit GameContractUpdated(newContract);
}

// EggVault
event NFTContractUpdated(address indexed newNFT);

function setEggNFT(address newNFT) external onlyOwner {
    require(newNFT != address(0), "Invalid address");
    eggNFT = newNFT;
    emit NFTContractUpdated(newNFT);
}

// EggHuntGame
event ThresholdUpdated(uint256 indexed newValue);

function setEggFindThreshold(uint256 newValue) external onlyOwner {
    require(newValue > 0, "Threshold must be positive");
    eggFindThreshold = newValue;
    emit ThresholdUpdated(newValue);
}
```

Adding these events ensures that critical administrative actions are logged and can be monitored on-chain, improving transparency and accountability.

[I-2] Public Function `EggVault::depositEgg(uint256 tokenId, address depositor)` Not Used Internally

Description:
The `depositEgg(uint256 tokenId, address depositor)` function is declared as `public` but is not utilized internally within the `EggVault` contract. This could indicate either an oversight or a potential design issue.

Impact:
Leaving unused public functions in the contract increases the attack surface and could lead to unintended usage or vulnerabilities. Additionally, it contributes to unnecessary code bloat, increasing deployment costs.

Proof of Concept:

```
function depositEgg(uint256 tokenId, address depositor) public {
    storedEggs[tokenId] = true;
    eggDepositors[tokenId] = depositor;
    require(eggNFT.ownerOf(tokenId) == address(this), "Invalid owner");
    emit EggDeposited(depositor, tokenId);
}
```

Recommended Mitigation:
1. If the function is not required, remove it to reduce the attack surface and deployment costs. 2. If the function is intended for external use, ensure it is properly documented and secured. 3. If the function is meant to be used internally, change its visibility to `internal` or `private` and refactor the code accordingly.

[I-3] Lack of Access Control in `EggVault::depositEgg(uint256 tokenId, address depositor)`

Description:
The `depositEgg(uint256 tokenId, address depositor)` function in the `EggVault` contract lacks access control, allowing any user to call it and deposit eggs on behalf of others. This could lead to unauthorized deposits, potentially causing confusion or misuse of the contract. If this isnt intended you should restrict access.

Impact:

Unauthorized users could manipulate the state of the contract by depositing eggs without proper permissions, leading to potential misuse or unexpected behavior.

Proof of Concept:

```
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);
}
```

Recommended Mitigation:
Restrict access to the `depositEgg` function by adding appropriate access control modifiers, such as `onlyContract` or a custom role-based modifier. For example:

```
function depositEgg(uint256 tokenId, address depositor) external
    onlyContract {
    require(eggNFT.ownerOf(tokenId) == address(this), "Invalid owner");
    storedEggs[tokenId] = true;
    eggDepositors[tokenId] = depositor;

    emit EggDeposited(depositor, tokenId);
}
```

Alternatively, implement a role-based access control mechanism using OpenZeppelin's `AccessControl` library to allow specific roles to call this function.

[I-4] Potential Reentrancy, State change after external call in `EggVault::depositEgg`

Description: The `EggHuntGame::depositEggToVault(uint256 tokenId)` function calls the `eggNFT.transferFrom()` (external call), then proceeds to call depositEgg(), which updates critical state variables. In this case, `EggstravaganzaNFT` is a trusted contract with no reentrant behavior and transferFrom() does not invoke external code. However, the pattern should be treated with caution.

Recommended Mitigation: Apply the nonReentrant modifier (from OpenZeppelin's ReentrancyGuard) on depositEgg() and withdrawEgg() for extra protection.

```
contract EggVault is Ownable, ReentrancyGuard {
    ...
    function depositEgg(...) public nonReentrant { ... }
}
```