puppy-raffle Audit Report

Version 1.0

yang

June 6, 2025

puppy-raffle Audit Report

yang

June 5, 2025

Lead Auditors: - yang

Table of Contents

- Table of Contents
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Protocol Summary
- Executive Summary
  - Issues found
- Findings
  - High
  - Medium
  - Low
  - Informational
  - Gas

Disclaimer

Our 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

```
./src/
-- PuppyRaffle.sol
```

Protocol Summary

The PuppyRaffle protocol is a decentralized NFT raffle system that allows users to participate in a lottery-style drawing for unique puppy NFTs. The protocol operates as follows:

1. **Raffle Entry**: Users can enter the raffle by paying an entry fee in ETH. Each entry grants them a chance to win a puppy NFT.

2. **Winner Selection**: After the raffle duration expires, the protocol automatically selects a winner using a random number generation mechanism. The winner receives a puppy NFT with randomly determined rarity.
3. **Fee Mechanism**: The protocol collects a percentage of the total entry fees, which can be withdrawn by the fee address.
4. **Refund System**: Users can withdraw refunds for their entries if they want.
5. **NFT Distribution**: The protocol mints and distributes puppy NFTs to winners, with each NFT having unique attributes and rarity levels.

The protocol aims to provide a fair and transparent way for users to participate in NFT raffles while maintaining a sustainable fee structure for protocol maintenance.

Roles

- Owner: Can change the fee address and manage protocol settings.
- Player: Can enter the raffle, request refunds, and potentially win the NFT.
- feeAddress: Can withdraw the fee in protocal.

Executive Summary

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	0
Info	3
Total	10

Findings

High

[H-1] Reentrancy in refund

Description:
The `refund` function implements an unsafe pattern by sending ETH to the user before updating the player's state, making it vulnerable to reentrancy attacks.

```
// src/PuppyRaffle.sol
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

        payable(msg.sender).sendValue(entranceFee);

        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact:
An attacker could exploit this vulnerability to repeatedly call `refund` and drain the contract's funds, potentially causing significant financial losses to the protocol.

Proof of Concept:
1. Attacker calls `refund` and receives ETH. 2. In the fallback function, attacker calls `refund` again before the player's state is updated. 3. This process repeats, draining the contract.

Recommended Mitigation:
Implement the Checks-Effects-Interactions pattern by updating the player's state before sending ETH, or use a reentrancy guard.

[H-2] Unsecure Randomness

Description:
The winner and rarity selection mechanism relies on `block.timestamp` and `block.difficulty` for randomness, which can be manipulated by miners.

```
// src/PuppyRaffle.sol
function selectWinner() external {
```

```
3       require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
4       require(players.length > 0, "PuppyRaffle: No players in raffle");
5
6       uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender
            , block.timestamp, block.difficulty))) % players.length;
7       address winner = players[winnerIndex];
8
9       // ... other code
10  }
```

Impact:
Miners can influence the outcome of the raffle by strategically including or excluding transactions, compromising the fairness of the protocol and potentially leading to unfair advantages.

Proof of Concept:
A miner can choose to include or exclude transactions to influence `block.timestamp` or `block.difficulty`, increasing their chance of winning.

Recommended Mitigation:
Use a secure randomness source, such as Chainlink VRF.

[H-3] Precision Loss in Fee Calculation

Description:
The `selectWinner` function uses integer division to calculate fees, which can lead to precision loss and incorrect fee distribution.

```
1  // src/PuppyRaffle.sol
2  function selectWinner() external {
3      // ... other code
4
5      uint256 prizePool = (totalAmountCollected * 80) / 100;
6      uint256 fee = (totalAmountCollected * 20) / 100;
7
8      // ... other code
9  }
```

Impact:
Due to Solidity's integer division truncation, the actual fees collected may be less than intended, potentially affecting protocol revenue and fund security.

Proof of Concept:

1. When calculating `feesToWithdraw = (totalAmountCollected * fee)/ 100`, if `totalAmountCollected * fee` is not perfectly divisible by 100, the remainder is

truncated.
2. For example, if `totalAmountCollected = 101` and `fee = 5`, the calculation would be `(101 * 5)/ 100 = 5.05`, but due to integer division, it becomes 5, losing 0.05 in fees.
3. This precision loss accumulates over multiple raffles, potentially resulting in significant revenue loss.

Recommended Mitigation:

1. Consider using a higher precision factor (multiply by 10000 instead of 100) to minimize precision loss.
2. Consider specifically designing a data type that uses integers to simulate floating-point numbers, which can minimize loss.

[H-4] Prize Distribution Failure to Contract Winners

Description:
The `selectWinner` function uses a low-level `call` to send the prize to the winner without checking if the recipient is a contract or has a fallback function.

```
// src/PuppyRaffle.sol
function selectWinner() external {
    // ... other code

    (bool success,) = winner.call{value: prizePool}(""); // <-- Unsafe call
    require(success, "PuppyRaffle: Failed to send prize pool to winner");

    // ... other code
}
```

Impact:
If the winner is a contract without a fallback function, the prize distribution will fail silently, potentially locking the prize funds in the contract forever.

Proof of Concept:

1. A contract without a fallback function wins the raffle
2. When `selectWinner` attempts to send the prize using `winner.call{value: prizePool}("")`, the transaction will fail
3. The prize funds remain locked in the contract as there is no mechanism to handle failed transfers
4. This affects both the prize pool and the contract's ability to continue operating

Recommended Mitigation:

1. Add a check to verify if the winner is a contract and has a fallback function before attempting the transfer
2. Implement a mechanism to handle failed transfers, such as:
    - Maintaining a record of failed transfers
    - Providing an alternative claiming mechanism for winners
    - Establishing a timeout period for unclaimed prizes
3. Consider using a more robust transfer mechanism that can handle both EOA and contract recipients

[H-5] Fee Withdrawal Condition Can Be Broken

Description:
The check `address(this).balance == totalFees` in `withdrawFees` can be broken by `selfdestruct`, potentially locking funds forever.

```
// src/PuppyRaffle.sol
function withdrawFees() external {
    require(address(this).balance == totalFees, "PuppyRaffle: There are currently players active!");
    // ... other code
}
```

Impact:
Funds may become permanently locked and unwithdrawable, directly affecting protocol revenue and fund security.

Proof of Concept:
Send ETH to the contract via `selfdestruct`, breaking the equality and preventing fee withdrawal.

Recommended Mitigation:
Check that `players.length == 0` instead, or implement a more robust mechanism to track active players.

[H-6] Fee Storage Type Limitation

Description:
The contract uses `uint64` type to store fees, which has a maximum value of 2^64-1 wei (approximately 18.44 ETH). This limitation could cause overflow issues if the total fees exceed this amount.

```
// src/PuppyRaffle.sol
uint64 public totalFees; // <-- Limited to 2^64-1 wei
```

Impact:
If the total fees collected exceed 2^64-1 wei, the contract will experience an integer overflow, potentially leading to incorrect fee calculations and fund loss. This is a direct risk to protocol funds and revenue.

Proof of Concept:

1. The `totalFees` variable is declared as `uint64`
2. Maximum value for uint64 is 2^64-1 (18,446,744,073,709,551,615 wei = 18.44 ETH)
3. If the protocol collects more than 18.44 ETH in fees, the `totalFees` variable will overflow
4. This overflow could lead to:
    - Incorrect fee calculations
    - Inability to track total fees accurately
    - Potential loss of funds due to overflow

Recommended Mitigation:

1. Change the `totalFees` variable type from `uint64` to `uint256`
2. Add explicit checks to ensure fee calculations don't exceed reasonable limits

Medium

[M-1] Duplicate Player Check is Inefficient and Prone to DoS

Description:
The duplicate check in `enterRaffle` implements an O(n^2) algorithm that can be exploited for DoS if the players array grows large.

```
// src/PuppyRaffle.sol
function enterRaffle() external payable {
    // ... other code ...

    for (uint256 i = 0; i < players.length - 1; i++) {
        for (uint256 j = i + 1; j < players.length; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate
                player");
        }
    }

    // ... other code
}
```

Impact:
A malicious user can fill the array, causing high gas costs and making the function uncallable.

Proof of Concept:
Call `enterRaffle` with a large number of addresses, causing the duplicate check to consume excessive gas.

Recommended Mitigation:
Implement a mapping to track player entries and prevent duplicates efficiently.

Informational / Non-Critical

[I-1] Address Zero Checks Missing

Description:
The constructor and `changeFeeAddress` do not check if the fee address is zero.

```
// src/PuppyRaffle.sol
constructor(address _feeAddress) {
    feeAddress = _feeAddress;
    // ... other code
}

function changeFeeAddress(address newFeeAddress) external onlyOwner {
    feeAddress = newFeeAddress;
    // ... other code
}
```

Impact:
Fees could be sent to the zero address and lost.

Recommended Mitigation:
Add a `require(newFeeAddress != address(0))` check.

[I-2] Inefficient Player Index Lookup

Description:
`getActivePlayerIndex` uses a linear search, a mapping could be more efficient.

```
// src/PuppyRaffle.sol
function getActivePlayerIndex(address player) external view returns (
    uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
```

```
9       }
```

Impact:
Gas costs increase as the number of players grows.

Recommended Mitigation:
Use a mapping from address to index.

[I-3] Function Visibility

Description:
Some functions could be marked `external` instead of **`public`** for clarity and gas savings.

Vulnerable Code:

```
// src/PuppyRaffle.sol
function getActivePlayerIndex() public view returns (uint256)
function getRaffleState() public view returns (RaffleState)
function getPlayerCount() public view returns (uint256)
```

Impact:
Slightly higher gas usage and less clear API.

Recommended Mitigation:
Mark functions as `external` where appropriate.

Gas

[G-1] Using `immutable` and `constant`

Description:
Variables like `raffleDuration`and image URIs could be marked as `immutable` or `constant` to save gas. Unnecessary gas usage for storage reads.

```
// src/PuppyRaffle.sol
uint256 public raffleDuration;
string public commonImageUri;
string public rareImageUri;
string public legendaryImageUri;
```

Recommended Mitigation:
Use `immutable` or `constant` for variables set only in the constructor or never changed.

[G-2] Unnecessary Variables

Description:
Variable `feesToWithdraw` is not needed and can be removed for gas savings.

Recommended Mitigation:
Remove unnecessary temporary variable.

[G-3] Unused Function and Inefficient Player Tracking

Description:
The `_isActivePlayer` function is never used in the contract and implements an inefficient O(n) lookup. The current implementation uses a linear search through the `players` array to check if an address is an active player.

Vulnerable Code:

```
// src/PuppyRaffle.sol
function _isActivePlayer() internal view returns (bool) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == msg.sender) {
            return true;
        }
    }
    return false;
}
```

Impact:
- Unused code increases contract size and deployment costs - Inefficient player tracking mechanism that could be optimized

Recommended Mitigation:

1. Remove the unused `_isActivePlayer` function
2. If player tracking is needed, implement a more efficient solution:

```
mapping(address => bool) public isPlayer;

// Update in enterRaffle
isPlayer[msg.sender] = true;

// Update in refund/selectWinner
isPlayer[player] = false;
```

3. This would reduce gas costs from O(n) to O(1) for player lookups