Puppy Raffle Audit Report

Version 0.1

April 16, 2025

Puppy Raffle Audit Report

Katlego Molokoane

April 16, 2025

Puppy Raffle Audit Report

Prepared by:

- Katlego Molokoane

Assisting Auditors:

- None

Table of contents

See table

- Issues found

- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance
    - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
    - [H-4] Malicious winner can forever halt the raffle
  - Medium
    - [M-1] Iterating through the `players` array to check for duplicates in the `PuppyRaffle::enterRaffle` function introduces a potential denial-of-service (DoS) vector. As the array grows, the gas costs for new entrants increase, making participation more expensive for future players.
    - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
    - [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    - [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
  - Informational / Non-Critical
    - [I-1] Floating pragmas
    - [I-2] Magic Numbers
    - [I-3] Test Coverage
    - [I-4] Zero address validation
    - [I-5] _isActivePlayer is never used and should be removed
    - [I-6] Unchanged variables should be constant or immutable
    - [I-7] Potentially erroneous active player index
    - [I-8] Zero address may be erroneously considered an active player
  - Gas

Disclaimer

The Katlego Molokoane team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

The CodeHawks severity matrix is utilized to dtermine the severity.

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Protocol Summary

Puppy Raffle is a decentralized application designed to facilitate the raffling of unique puppy-themed NFTs with varying levels of rarity (e.g., common, rare, legendary). Participants can enter the raffle by paying an entrance fee, which contributes to the prize pool. At the end of each raffle, a winner is selected to receive the prize pool and a randomly assigned NFT.

The protocol includes the following key features: - **Fee Mechanism**: A portion of the entrance fees is allocated to a designated `feeAddress`, controlled by the protocol owner. - **Randomness**: The winner and the rarity of the NFT are determined using on-chain randomness.

The protocol aims to provide a fair and transparent mechanism for distributing NFTs while ensuring a portion of the fees is allocated to the protocol's maintenance or other purposes defined by the owner.

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	4
Medium	4
Low	0
Info	8
Gas	3
Total	19

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, an external call is made to the `msg.sender` address before updating the `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 implement a `fallback` or `receive` function in their contract to repeatedly invoke the `PuppyRaffle::refund` function. This would allow them to claim multiple refunds in a loop, potentially draining the entire contract balance.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. A user enters the raffle.
2. An attacker deploys a malicious contract with a `fallback` function designed to invoke the `PuppyRaffle::refund` function.
3. The attacker uses their malicious contract to enter the raffle.
4. The attacker repeatedly calls the `PuppyRaffle::refund` function through their contract, exploiting the reentrancy vulnerability to drain the contract's balance.

Proof of Code:

Code

Add the following code to the `PuppyRaffleTest.t.sol` file.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(address _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);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}

function testReentrance() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(address(puppyRaffle));
    vm.deal(address(attacker), 1e18);
    uint256 startingAttackerBalance = address(attacker).balance;
    uint256 startingContractBalance = address(puppyRaffle).balance;

    attacker.attack();

    uint256 endingAttackerBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;
    assertEq(endingAttackerBalance, startingAttackerBalance + startingContractBalance);
    assertEq(endingContractBalance, 0);
}
```

Recommended Mitigation: The `PuppyRaffle::refund` function should update the `players` array before making any external calls to ensure state consistency and mitigate reentrancy risks. Additionally, the event emission should be moved prior to the external call to accurately reflect the state changes.

```
    function refund(uint256 playerIndex) public {
```

```diff
         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);
         (bool success,) = msg.sender.call{value: entranceFee}("");
         require(success, "PuppyRaffle: Failed to refund player");
-         players[playerIndex] = address(0);
-         emit RaffleRefunded(playerAddress);
     }
```

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

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together results in a predictable value. This approach does not provide sufficient randomness, as these inputs can be manipulated or anticipated by malicious users, allowing them to influence the outcome and select the raffle winner.

Impact: Any participant can manipulate the process to select the raffle winner, enabling them to claim the prize money and choose the "rarest" puppy. This effectively undermines the rarity distribution of the puppies, as the ability to select the winner allows all puppies to be treated as having equal rarity.

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.

```
uint64 myVar = type(uint64).max;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In the `PuppyRaffle::selectWinner` function, the `totalFees` variable is used to accumulate fees for the `feeAddress`, which can later be withdrawn using the `withdrawFees` function. However, if the `totalFees` variable overflows, the `feeAddress` may not receive the correct amount of fees, potentially leaving the excess fees permanently locked in the contract.

Proof of Concept: 1. A raffle is concluded with 4 players, resulting in the collection of some fees.
2. Subsequently, 89 additional players enter a new raffle, which is also concluded.
3. The `totalFees` variable is updated as follows:
`javascript totalFees = totalFees + uint64(fee); //Substituted values : totalFees = 800000000000000000 + 17800000000000000000; // Due to overflow, the resulting value becomes: totalFees = 153255926290448384;`
4. As a result of the overflow, withdrawals are blocked by the following line in the `PuppyRaffle::withdrawFees` function:
`javascript require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`

While it is technically possible to use `selfdestruct` to send ETH to the contract and align the balances to enable withdrawals, this approach is clearly not aligned with the intended functionality of the protocol.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
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

        // 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);
        // End the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        //  The issue occurs here.
        puppyRaffle.selectWinner();

```

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

        // 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();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- 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`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

```
- 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.

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

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

If the `winner` account is a smart contract that does not implement a payable `fallback` or `receive` function, or if these functions are implemented but revert, the external call to transfer the prize will fail. This failure will cause the execution of the `selectWinner` function to halt, preventing the prize from being distributed and blocking the raffle from starting a new round.

Additionally, another attack vector exists that can halt the raffle. The `selectWinner` function mints an NFT to the winner using the `_safeMint` function, which is inherited from the `ERC721` contract. This function attempts to call the `onERC721Received` hook on the recipient if it is a smart contract. If the recipient contract does not implement the `onERC721Received` hook, the minting process will revert, causing the `selectWinner` function to fail.

As a result, an attacker can register a smart contract in the raffle that does not implement the required `onERC721Received` hook. This will prevent the NFT from being minted and will cause the `selectWinner` function to revert, effectively halting the raffle.

Impact: In either scenario, the inability to distribute the prize would prevent the raffle from starting a new round, effectively halting the raffle indefinitely.

Proof of Concept:

Proof Of Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
```

```
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Medium

[M-1] Iterating through the `players` array to check for duplicates in the `PuppyRaffle::enterRaffle` function introduces a potential denial-of-service (DoS) vector. As the array grows, the gas costs for new entrants increase, making participation more expensive for future players.

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 that 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.

Impact:

The gas costs for raffle entrants will greatly increase as more players enter the raffle.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252039 - 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
        // Check for duplicates
@>      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");
            }
        }
```

Proof Of Code

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

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

        // We will enter 5 players into the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // And see how much gas it cost to enter
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

        // We will enter 5 more players into the raffle
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        // And see how much more expensive it is
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        gasEnd = gasleft();
        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
        // Logs:
        //     Gas cost of the 1st 100 players: 6252039
        //     Gas cost of the 2nd 100 players: 18067741
}
```

Recommended Mitigation: There are a few recommended mitigations.

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.

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

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

[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 verifies that `totalFees` matches the contract's ETH balance (`address(this).balance`). Although the contract lacks a `payable` fallback or `receive` function, a user could `selfdestruct` a contract containing ETH to forcibly send funds to the `PuppyRaffle` contract, bypassing this check.

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

```
    }
```

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.

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

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In the `PuppyRaffle::selectWinner` function, a `uint256` is cast to a `uint64`. This is an unsafe operation, as any value exceeding `type(uint64).max` will be truncated during the cast.

```
    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:

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:

```
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. There 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");
        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-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 owness on the winner to claim their prize. (Recommended)

Informational / Non-Critical

[I-1] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[I-3] 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.

```
| 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% (20/30) | 77.78% (7/9)   |
| 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% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
    PuppyRaffle.sol#57) lacks a zero-check on :
                    - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
    sol#165) lacks a zero-check on :
```

```
                    - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the `feeAddress` is updated.

[I-5] _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;
-    }
```

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be
    constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[I-7] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

Gas

- `getActivePlayerIndex` returning 0: Does this indicate the player is at index 0, or that the player is inactive?
- randomness for rarity issue
- Reentrancy in `PuppyRaffle` before `_safeMint`