Protocol Audit Report

Version 1.0

Tim Sigl

June 14, 2024

Protocol Audit Report

Tim Sigl

2024-06-14

Prepared by: Tim Sigl Lead Security Researcher:

- Tim Sigl

Table of Contents

Protocol Summary

The PuppyRaffle contract facilitates a raffle for winning unique puppy-themed NFTs. Participants enter the raffle by paying an entrance fee, with duplicate entries being prohibited. The contract selects a winner periodically based on a random selection algorithm, distributing 80% of the total fees collected to the winner and the remaining 20% to a designated fee address. Users can request refunds for their entry fees if they meet specific criteria. The contract also supports changing the fee address and withdrawing accumulated fees. Metadata for the NFTs, including rarity and image URIs, is generated dynamically. Overall, the contract aims to provide a fair and transparent mechanism for users to participate in and win puppy-themed NFTs through a blockchain-based raffle system.

Disclaimer

The Tim-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

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

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

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

The review was conducted on the PuppyRaffle contract to identify security vulnerabilities and potential issues. The audit focused on the contract's functionality, security, and best practices. The audit identified several high, medium, and low severity issues that could impact the security, integrity, and usability of the contract. The findings are summarized below:

Issues found

Severity	Number of Findings
High	3
Medium	5
Low	2
Info	7
Total	5

Findings

High

[H-1] Reentrancy vulnerability in `PuppyRaffle::refund` due to state update after external call

Description:

The `PuppyRaffle::refund` function inappropriately updates the contract state after transferring funds to `msg.sender`. This allows for reentrancy attacks where a malicious contract can call back into the refund function before state changes are fully applied, potentially draining all funds from the contract.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 96-105

```
function refund(uint256 playerIndex) public {
    ...
    payable(msg.sender).sendValue(entranceFee);
    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

Impact: By updating the state after interacting with msg.sender, the function enables a reentrancy vulnerability. This allows an attacker to repeatedly call the refund function, potentially draining all funds from the contract if not properly guarded against.

Proof of Concept:

In the following example the `ReentrancyAttacker` is the contract attacking the `PuppyRaffle::refund` function. The attacker enters the raffle and then refunds the entrance fee. The attacker then calls the `refund` function again in the `receive` function, which is called when the contract receives funds. This allows the attacker to drain the contract of all funds.

The second code snippet is a test that demonstrates the reentrancy attack.

Test code snippet

Attacker:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

```

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

Attack demonstration test:

```
1   function test_reentrancyRefund() public {
2           address[] memory player = new address[](4);
3           player[0] = playerOne;
4           player[1] = playerTwo;
5           player[2] = playerThree;
6           player[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(player);
8            assert(address(puppyRaffle).balance == entranceFee * 4);
9
10          // Attack
11          ReentrancyAttacker attacker = new ReentrancyAttacker(
                puppyRaffle);
12          address attackUser = makeAddr("ATTACKER");
13          uint256 startingBalance = 1 ether;
14          vm.deal(attackUser, startingBalance);
15          console.log("Attacker starting balance: ", startingBalance);
16          console.log("PuppyRaffle starting balance: ", address(
                puppyRaffle).balance);
17
18          vm.prank(attackUser);
19          attacker.attack{value: entranceFee}();
20          console.log("Attacker balance after attack: ", address(attacker
                ).balance);
21          console.log("PuppyRaffle balance: ", address(puppyRaffle).
                balance);
22      }
```

Recommended Mitigation:

Ensure that state changes are made after interacting with external contracts to prevent reentrancy attacks. One way to do this is to use the “Checks-Effects-Interactions” pattern, where state changes are made after all external interactions are complete. Move the state update and the event emission above the external call to `sendValue`.

The other mitigation strategy is to use the OpenZeppelin ReentrancyGuard library to protect against reentrancy attacks. The library provides a modifier that can be added to functions to prevent reentrancy.

Mitigation Example

```
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");

    // Transfer funds after state changes to prevent reentrancy
+    players[playerIndex] = address(0);
+    emit RaffleRefunded(playerAddress);

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` function can be exploited to predict the winner and the rarity of the NFT

Description: The `selectWinner` function utilizes weak randomness generation through keccak256 of `msg.sender`, `block.timestamp`, and `block.difficulty` to determine the winner of the raffle. This method of randomness is insecure for selecting winners in a raffle, as it can be predicted or manipulated by attackers.

Another attack vector here is frontrunning the `selectWinner` transaction to quickly refund the entrance fee in the case the user is not the selected winner.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 125-154

```
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

```
3       require(players.length >= 4, "PuppyRaffle: Need at least 4 players"
            );
4       uint256 winnerIndex =
5           uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
                block.difficulty))) % players.length;
6       address winner = players[winnerIndex];
7       uint256 totalAmountCollected = players.length * entranceFee;
8       uint256 prizePool = (totalAmountCollected * 80) / 100;
9       uint256 fee = (totalAmountCollected * 20) / 100;
10      totalFees = totalFees + uint64(fee);
11
12      uint256 tokenId = totalSupply();
13
14      // We use a different RNG calculate from the winnerIndex to
            determine rarity
15      uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
            block.difficulty))) % 100;
16      if (rarity <= COMMON_RARITY) {
17          tokenIdToRarity[tokenId] = COMMON_RARITY;
18      } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
19          tokenIdToRarity[tokenId] = RARE_RARITY;
20      } else {
21          tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
22      }
23  }
```

Impact: Weak randomness allows attackers to potentially predict or influence the outcome of the raffle, compromising the fairness and integrity of the contract. This vulnerability undermines the core functionality of selecting a winner randomly and could lead to financial loss and loss of trust among participants.

Proof of Concept:

- Validators know the `block.timestamp` and `block.difficulty` before the block is mined. This allows them to predict the winner of the raffle and the rarity of the NFT.
- Users can generate addresses to change `msg.sender` until they are selected as the winner.
- Users can simply revert the transaction if they are not selected as the winner or the rarity is not what they want.

Recommended Mitigation: Use a secure source of randomness to select the winner of the raffle. One option is to use Chainlink VRF (Verifiable Random Function) to generate a random number that cannot be predicted or manipulated by any party. Chainlink VRF provides a provably fair and tamper-proof source of randomness that can be used to ensure the integrity of the raffle.

[H-3] Integer overflow of `PuppyRaffle::totalFees` results in loss of fees

Description: The `totalFees` variable is of type `uint64` and can overflow if the sum of fees exceeds the maximum value of `uint64`. This can result in a loss of fees and incorrect accounting of the total fees collected by the contract. In Solidity versions prior to 0.8.0, integers are susceptible to overflows.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 134

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

Impact:

If the `totalFees` variable overflows, the protocol will misreport the collected fees, resulting in potential financial losses and incorrect contract behavior. Specifically, the `feeAddress` may not collect the correct amount of fees, and funds could remain trapped in the contract.

Proof of Concept:

1. Conclude a raffle of 4 players to collect some fees.
2. Have 89 additional players enter a new raffle, then conclude that raffle as well.

Calculation of `totalFees`:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted values
3 totalFees = 800000000000000000 + 17800000000000000000;
4 // due to overflow, totalFees will be incorrect
5 totalFees = 153255926290448384;
```

Attempting to withdraw fees will fail due to the require check in `PuppyRaffle::withdrawFees`:

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

Proof of code:

Test code snippet

```
1 function testTotalFeesOverflow() public {
2     // We finish a raffle of 4 to collect some fees
3     vm.warp(block.timestamp + duration + 1);
4     vm.roll(block.number + 1);
5     puppyRaffle.selectWinner();
6     uint256 startingTotalFees = puppyRaffle.totalFees();
7     // startingTotalFees = 800000000000000000
8
9     // We then have 89 players enter a new raffle
```

```
10      uint256 playersNum = 89;
11      address[] memory players = new address[](playersNum);
12      for (uint256 i = 0; i < playersNum; i++) {
13          players[i] = address(i);
14      }
15      puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
16      // We end the raffle
17      vm.warp(block.timestamp + duration + 1);
18      vm.roll(block.number + 1);
19
20      // Issue occurs here
21      // We will now have fewer fees even though we just finished a
            second raffle
22      puppyRaffle.selectWinner();
23
24      uint256 endingTotalFees = puppyRaffle.totalFees();
25      console.log("ending total fees", endingTotalFees);
26      assert(endingTotalFees < startingTotalFees);
27
28      // Unable to withdraw any fees due to the require check
29      vm.prank(puppyRaffle.feeAddress());
30      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
31      puppyRaffle.withdrawFees();
32  }
```

Recommended Mitigation:

- Use a Solidity version greater than 0.8.0, which includes built-in checks for integer overflows.
- In older versions use the SafeMath library to prevent integer overflows and underflows.
- Consider using a larger integer type for `totalFees` to prevent overflow.
- Remove or modify the balance check in `PuppyRaffle::withdrawFees`.

Medium

[M-1]: Denial-of-Service in `PuppyRaffle::enterRaffle`, unbound variable `player` can exceed block gas limit eventually denying execution of the function

Description: There is a possibility for denial of service (DoS) due to a gas inefficient double loop in the contract. The nested loops iterate over the players array, and the gas costs scale linearly with the number of players. This inefficiency could allow an attacker to create multiple accounts, increasing gas costs and potentially reaching the block gas limit, thus denying execution of the contract.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 86-87

```
        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");
            }
        }
```

Proof of Concept: The following tests shows that the gas costs grow linearly with the number of players:

Test code snippet

```
function testDosEnterRaffle() public {
        uint256 newPlayersToJoin = 100;
        for (uint256 i = 0; i < newPlayersToJoin; i++) {
            address[] memory players = new address[](1);
            players[0] = address(uint160(i));
            uint256 gasBefore = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee}(players);
            uint256 gasAfter = gasleft();
            console.log("Gas used:", gasBefore - gasAfter);
        }
    }
```

Recommended Mitigation: There are two potential options:

1. Remove the check for duplicate address since players could enter with other wallets multiple times either way
2. Store the players that have entered in a mapping which can directly accessed by player address for checking if the player has already entered
    1. OpenZeppelins enumerables might help here: Enumerables

Impact: The gas inefficiency in the double loop can lead to high transaction costs and potential denial of service if an attacker exploits the inefficiency by creating a large number of players.

Recommended Mitigation:

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: The `fee` variable is cast to `uint64` before being added to the `totalFees` variable. If the `fee` value exceeds the maximum value of `uint64`, the fees will be lost due to overflow. This can lead to incorrect accounting of the total fees collected by the contract.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 134

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

Impact:

If the `fee` variable exceeds the maximum value of `uint64`, the fees will be lost due to overflow. This can lead to incorrect accounting of the total fees collected by the contract and potential financial losses.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the fee as a uint64 hits
3. totalFees is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// outputs 0
```

Recommended Mitigation:

Ensure that the `totalFees` variable can accommodate the sum of all fees collected by the contract. Consider using a larger integer type for `totalFees` to prevent overflow and remove the cast to `uint64`.

[M-4] Smart contract wallet winners without a receive or fallback block new contest and payouts

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, the lottery cannot be reset, preventing a new contest from starting.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact:

- The `PuppyRaffle::selectWinner` function could revert many times, making it difficult to reset the lottery and preventing new ones from starting.
- True winners would not be able to get paid out, and someone else could win their money.

Proof of Concept:

1. 10 smart contract wallets without a fallback or receive function enter the lottery.

2. The lottery ends.
3. The `selectWinner` function fails to execute, preventing the lottery from resetting.

Recommended Mitigation:

- You could prevent smart contract wallets from entering the lottery. However, this also prevents multisig wallets from participating and is therefore not recommended.
- The better option is to use a pull payment mechanism, where the winner can withdraw their winnings instead of having them sent automatically.

[M-5] Forced ether send to the contract prevents `PuppyRaffle::withdrawFees` from withdrawing fees

Description:

In Ethereum, contracts can refuse to accept Ether by not implementing a `receive` or `fallback` function, or by having a receive function that reverts. However, the `selfdestruct` function in Solidity allows a contract to send its balance to any address, including a contract that normally refuses to accept Ether. This action cannot be prevented by the target contract.

Because of the

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

in the `PuppyRaffle::withdrawFees` function, the contract will not be able to withdraw the fees because the contracts balance will not be equal to the total fees.

Impact: This would prevent the feeAddress from withdrawing fees. A malicious user could see a withdrawFee transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. PuppyRaffle has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a selfdestruct
3. `feeAddress` is no longer able to withdraw funds because of the require check in `PuppyRaffle::withdrawFees`

Recommended Mitigation:

- Remove the balance check in `PuppyRaffle::withdrawFees` and allow the `feeAddress` to withdraw the fees regardless of the contract's balance.

Low

[L-1] Confusing behavior in `PuppyRaffle::getActivePlayerIndex` function returning 0 when player is not found

Description: The `PuppyRaffle::getActivePlayerIndex` function returns index 0 when a player is not found in the `players` array. This can lead to confusion because the index 0 is also returned when the first player who enters and is active is queried, potentially leading to misinterpretation of whether the player is active or not.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 110-117

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

Impact: This issue introduces confusion in determining whether a player is active or not, especially when the index 0 is returned for both non-existent players and the first active player. This confusion can affect the logic and user interface of applications using this function.

Proof of Concept:

1. The first player enters the raffle and is assigned index 0.
2. The player checks if they are active using the `getActivePlayerIndex` function.
3. The function returns 0, indicating that the player is not active, which is misleading.
4. The player joins the raffle again, because they think they have not entered yet.

Recommended Mitigation: There are two possible solutions to this issue:

1. Revert with an error message when the player is not found in the `players` array.
2. Return a value that cannot be confused with an active player index, such as `-1`.

[L-2] Stuck funds in the contract due to lost precision `prizePool` and `fee` calculations

Description: Solidity does not support floating point arithmetic and cuts of decimal places when dividing. Although `prizePool` and `fee` are first multiplied by a percentage to minimize precision

loss, there is still a small amount of precision lost as shown in the PoC. Because the `PuppyRaffle:withdraw` function does not withdraw the funds from the contracts balance but rather from the `totalFees` variable, the precision loss can lead to funds being stuck in the contract.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 132-133

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

Impact: The precision loss in the `prizePool` and `fee` calculations can lead to small amounts of funds being stuck in the contract. Over time, these funds can accumulate and potentially become a significant amount, affecting the contract's financial health and usability.

Proof of Concept: The following test demonstrates the precision loss in the `prizePool` and `fee` calculations: Example:

1. The `totalAmountCollected` is 1234.
2. 1234*80=98720
3. 98720/100=987.2
4. The .2 is cut of and stuck in the contract.

Recommended Mitigation:

1. Scale the `totalAmountCollected` to a higher precision before performing the percentage calculations.
2. Change the `withdraw` function to withdraw the funds from the contract balance instead of the `totalFees` variable.

Informational

[I-1] Solidity pragma should be specific, not wide, different compiler versions can cause unforeseen issues

Description:

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

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

Impact: Using a different version of Solidity to compile the production contract can cause unforeseen bugs.

Recommended Mitigation:

Change to specific Solidity version:

```
1 -   pragma solidity ^0.7.6;
2 +   pragma solidity 0.8.26;
```

[I-2] Old Solidity version in use, using an old version prevents access to new Solidity security checks

Description:

Solidity version `0.7.6` is in use. If possible upgrade to a recent version of Solidity (at least 0.8.0) with no known severe issues.

Impact:

Using an old version prevents access to new Solidity security checks. Can result in less secure code.

Recommended Mitigation:

Change to a recent version of Solidity.

```
1 -   pragma solidity ^0.7.6;
2 +   pragma solidity 0.8.26;
```

[I-3] Missing checks for address(0) when assigning values to address state variables

Description:

The contract does not check for address(0) (the zero address) when assigning values to address state variables. This omission can lead to unintended behaviors if the zero address is inadvertently assigned.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1     constructor(uint256 _entranceFee, address _feeAddress, uint256
          _raffleDuration) ERC721("Puppy Raffle", "PR") {
```

```
2           entranceFee = _entranceFee;
3           feeAddress = _feeAddress;
4       ...
5       }
```

Impact: Assigning the zero address (address(0)) to address state variables can lead to funds being irrecoverably lost or unintended control flows in the contract. No in this case since the address can be changed by the owner.

Recommended Mitigation:

Mitigation Example

```
1 -         feeAddress = _feeAddress;
2 +         require(_feeAddress != address(0), "Fee address cannot be zero
      ");
3 +         feeAddress = _feeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, thus not following best practices

Description: The function calls `_safeMint` after sending funds to the winner (interaction). This violates the Checks-Effects-Interactions pattern, which can lead to reentrancy vulnerabilities. In this specific case reentrancy is not possible because checks are done in the beginning of the function, but it is still a good practice to follow CEI.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 125-154

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length >= 4, "PuppyRaffle: Need at least 4
                players");
4           uint256 winnerIndex =
5               uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
6           address winner = players[winnerIndex];
7           uint256 totalAmountCollected = players.length * entranceFee;
8           uint256 prizePool = (totalAmountCollected * 80) / 100;
9           uint256 fee = (totalAmountCollected * 20) / 100;
10          totalFees = totalFees + uint64(fee);
11
12          uint256 tokenId = totalSupply();
13
14          // We use a different RNG calculate from the winnerIndex to
                determine rarity
```

```
15          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
                block.difficulty))) % 100;
16          if (rarity <= COMMON_RARITY) {
17              tokenIdToRarity[tokenId] = COMMON_RARITY;
18          } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
19              tokenIdToRarity[tokenId] = RARE_RARITY;
20          } else {
21              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
22          }
23
24          delete players;
25          raffleStartTime = block.timestamp;
26          previousWinner = winner;
27          (bool success,) = winner.call{value: prizePool}("");
28          require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
29          _safeMint(winner, tokenId);
30      }
```

Impact: Although reentrancy is not possible in this case, following the CEI pattern is a best practice to prevent reentrancy vulnerabilities. By following CEI, the contract can be more secure and less prone to reentrancy attacks in other parts of the code.

Recommended Mitigation:

Move the `_safeMint` function call above the external call to `winner.call{value: prizePool}("")` to follow the CEI pattern.

Mitigation Example

```
1           delete players;
2           raffleStartTime = block.timestamp;
3           previousWinner = winner;
4   +        _safeMint(winner, tokenId);
5           (bool success,) = winner.call{value: prizePool}("");
6           require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
7   -       _safeMint(winner, tokenId);
```

[I-5] Use of Magic Numbers in `prizePool` and `fee` calculations make the code less readable and maintainable

Description: The code uses magic numbers (`80`, `20`, and `100`) directly in calculations for `prizePool` and `fee` percentages, which reduces readability, maintainability, and may lead to errors when modifying or understanding the code in the future.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 132-133

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

Impact:

Using magic numbers obscures the meaning of these constants (`80`, `20`, and `100`) and makes it challenging to update or adjust these values consistently throughout the codebase. This practice can lead to unintended behavior, such as incorrect calculations or misinterpretations during code maintenance or auditing.

Recommended Mitigation:

Define constants for the magic numbers to improve code readability and maintainability. This approach makes the code self-explanatory and easier to modify in the future.

Mitigation Example

```
1  // Define constants for clarity and reusability
2  +   uint256 constant PRIZE_POOL_PERCENTAGE = 80;
3  +   uint256 constant FEE_PERCENTAGE = 20;
4  +   uint256 constant PRIZE_PRECISION = 100;
5
6  function distributeFunds() internal {
7      ...
8      // Calculate prize pool and fee using constants
9  +   uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
       / PRIZE_PRECISION;
10 +   uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
       PRIZE_PRECISION;
11     ...
12 }
```

[I-6] Missing event emission for storage updates

Description: Some functions in the contract update the storage but do not emit events. Emitting events when storage is updated is important for transparency and traceability, as it allows off-chain tools and external observers to track state changes within the contract.

Example:

The refund function updates the players array but does not emit an event for this update. This makes it difficult to track which players have been refunded.

Code Snippet

```
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);  // Event emitted for refund
}
```

Impact: Without event emissions for storage updates:

- External observers may not be aware of critical state changes, reducing the transparency of the contract.
- Off-chain tools and dApps relying on events to track contract state may not function correctly, leading to potential discrepancies and trust issues.
- Auditing and debugging become more difficult, as there is no easy way to track changes to the contract's state.

Recommended Mitigation:

Ensure that all functions updating the contract's storage emit appropriate events to reflect these changes.

Mitigation Example

Add an event emission for the players array update in the refund 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);  // Emitting event for refund
}
```

[I-7] Remove unused function `PuppyRaffle::_isActivePlayer`

Description: The `_isActivePlayer` function is defined but not used in the contract. Removing unused functions helps to reduce the contract's complexity and makes the codebase cleaner and easier to maintain.

Impact: Having unused functions in the contract adds unnecessary complexity and can confuse developers and auditors. It is best practice to remove unused functions to improve code readability and maintainability.

Recommended Mitigation:

Remove the unused `_isActivePlayer` function from the contract.

Gas

[G-1] Variables are not declared immutable, leading to increased gas costs

Description: The variables in the contract that are only set during the constructor and not changed afterward are not declared as immutable. Declaring these variables as immutable can save gas during contract execution.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 24-25

```
1    uint256 public raffleDuration;
2    uint256 public raffleStartTime;
```

Impact: Using immutable for these variables reduces gas costs, making the contract more efficient and cost-effective. It ensures that the variables are stored directly in the bytecode, leading to savings on storage operations.

Recommended Mitigation:

Mitigation Example

```
1  -    uint256 public raffleDuration;
2  -    uint256 public raffleStartTime;
3  +    uint256 public immutable raffleDuration;
4  +    uint256 public immutable raffleStartTime;
```

[G-2] Variables are not declared constant, leading to increased gas costs

Description: The variables `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri`, and `PuppyRaffle::legendaryImageUri` are assigned fixed values and never modified. These variables could be declared as `constant` to optimize gas usage.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 38

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

- Found in src/PuppyRaffle.sol Line: 43

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

- Found in src/PuppyRaffle.sol Line: 48

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

Impact: Not declaring these variables as `constant` leads to higher gas costs during contract execution since the values are stored in storage instead of directly in the bytecode.

Recommended Mitigation:

Mitigation Example

```
-     string private commonImageUri = "ipfs://
    QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
-     string private rareImageUri = "ipfs://
    QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
-     string private legendaryImageUri = "ipfs://
    QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
+     string private constant commonImageUri = "ipfs://
    QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
+     string private constant rareImageUri = "ipfs://
    QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
+     string private constant legendaryImageUri = "ipfs://
    QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
```

[G-3] Event is missing indexed fields, may result in lower performance when parsing events

Description: Indexing event fields allows them to be quickly accessible to off-chain tools that parse events. Each indexed field increases gas costs during event emission, so indexing should be used judiciously based on the needs of the application.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 53-55

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

Impact: Events without indexed fields may result in slower event data processing for off-chain applications. Indexed fields enhance the efficiency of event log querying and filtering.

Recommended Mitigation:

Mitigation Example

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

[G-4] Gas optimization: Cache `PuppyRaffle::players` length in a variable

Description: The length of `PuppyRaffle::players` is accessed multiple times in a loop without caching its value in a variable. Caching the length in a variable improves gas efficiency by avoiding redundant length calculations on each iteration.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 86-87

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

Impact: Redundant length calculations in loops can lead to increased gas costs, especially in loops iterating over arrays or dynamic lists.

Recommended Mitigation:

Mitigation Example

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

[G-5] Event emitted even when newPlayers array is empty in `PuppyRaffle:enterRaffle`

Description: The enterRaffle function emits the RaffleEnter event regardless of whether the newPlayers array is empty or not. This results in unnecessary event emission when no new players actually entered the raffle.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 91

```
     emit RaffleEnter(newPlayers);
```

Impact: Event emission consumes gas and adds unnecessary data to the blockchain when no new players are added to the raffle. This can lead to increased transaction costs and inefficient use of blockchain resources.

Recommended Mitigation: Add a check to emit the event only when the newPlayers array is not empty.

Mitigation Example

```
+ if (newPlayers.length > 0) {
+     emit RaffleEnter(newPlayers);
+ }
```