Protocol Audit Report

Prepared by: Fish Scale

Table of Contents

Protocol Summary

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

Disclaimer

The Fish Scale 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

		Impact		
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

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

Issues found

Findings

Severity	Number of Issues Found
High	3
Medium	1
Low	0
Info	0

Total 4

High

[H-2] Improper uasge of code was written in attempt to produce a viable RNG, allow bad actors to predict and manipulate this protocol

Description: Code was written as an attempt to produce a truly viable RNG system for this contract.

Impact: explain whats wrong

Proof of Concept:

```
        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;
        }
```

Recommended Mitigation: Recommend deploying and usage of Chainlink

Liklihood & Impact

- Impact: High
- Liklihood: High
- Severity: High

[H-3] Function fails to enforce CEI principles, leaving it vulnerable to reentrancy attacks

Description: The `PuppyRaffle::withdrawFee` failed to enforce CEI principle as it failed to assure an address is not a 0 address.

Impact: function vulnarable to reentrancy attack

Proof of Concept:

```
    function withdrawFees() external {
        require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Recommended Mitigation:

```
    function withdrawFees() external {
        require(address(this).balance == uint256(totalFees), "PuppyRaffle:
There are currently players active!");
+     require(address != address(0), "PuppyRaffle: Cannot withdraw to 0
address");
```

```
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Liklihood & Impact

- Impact: High
- Liklihood: High
- Severity: High

Medium

[M-1] Looping through `PuppyRaffle::enterRaffle` for duplicates lead to a DOS Attack. Denial Of Service Attack

Description: The code in `PuppyRaffle::enterRaffle`to loop for duplicates is a major error as it may lead to an attacker to spam the function and dramatically increase the gas fees for the next onboarding `player`.

Impact: A dedicated hacker can and will spam the `PuppyRaffle::enterRaffle` function to harm the protocol by cheating onboarding users into paying a signifcant amount of gas. An inflated gas cost discourages ```players`` form onboardiong this protocol ensuring the hacker with higher odds to win then raffle. This operation is an expensive one yet, a great deal of a threat.

Proof of Concept:

1st set of players = 100 2nd set of player = 200

-gas cost set 1 ~6252128

-gas cost set 2 ~18068219

▶ POC

```
javascript
 function test_denial_of_service() public {
        vm.txGasPrice(1);

        uint  playerNum = 100;
        address[] memory players= new address[](playerNum);
        for(uint i =0; i< playerNum; i++){
            players[i] = address(i);
        }
        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint gasEnd = gasleft();
        uint gasUsed = (gasStart - gasEnd) * tx.gasprice;
```

```
        console.log("Gas used for entering %s players: %s", playerNum,
gasUsed);

                vm.txGasPrice(1);

        uint  playerNumto = 100;
        address[] memory playerso = new address[](playerNumto);
        for(uint i =0; i< playerNumto; i++){
            playerso[i] = address(i + playerNumto);
        }
        uint gasStarts = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerso.length}
(playerso);
        uint gasEnds = gasleft();
        uint gasUseds= (gasStarts - gasEnds) * tx.gasprice;
        console.log("Gas used for entering %s players: %s", playerNumto,
gasUseds);

    assert(gasUseds > gasUsed); // this is a denial of service attack
    }
```

Recommended Mitigation: Use a mapping type to evaluate if there are any duplicated players. Or Use OppenZepplin EnumerableSet to examine any player duplications

```
```

Liklihood & Impact

- Impact: MEDIUM
- Liklihood: MEDIUM
- Severity: MEDIUM