Audit Report - Puppy Raffle

Title: **Puppy Raffle Audit Report**

Author: **DHANANJAY BHAVAR**

Date: **January 3, 2023**

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following: Call the `enterRaffle` function with the following parameters:

`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.

Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and mint a random puppy 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.

Risk Classification

`High`: Direct impact on the funds or the main functionality of the protocol.

`Medium`: Indirect impact on the funds or the protocol's functionality.

`Low`: Minimal to no impact on the funds or the protocol's main functionality.

`Informational`: Suggestions related to good coding practices and gas-efficient code.

Audit Scope

`commit hash`: e30d199697bbc822b646d76533b66b7d529b8ef5

`contracts`: ./src/PuppyRaffle.sol

`Solc Version`: 0.7.6

`Chain(s)`: Ethereum

Findings Summary

Severity	No of issues found
HIGH	2
MEDIUM	4
LOW	1
INFO	6
TOTAL	13

FINDINGS

[H-1] Re-Entrancy Vulnerability in refund Function

Description: The refund function may be susceptible to a Re-Entrancy attack. In its current state, the function sends funds to `msg.sender` before updating the players array. This order of operations could allow a malicious player to repeatedly invoke the refund function and re-enter it before the state changes take effect.

Impact: A malicious player could exploit the vulnerability to repeatedly trigger the refund function, causing unexpected behavior and potentially draining the contract's funds.

Recommended Mitigation: To mitigate the risk of re-entrancy attacks, consider changing the order of operations and follow the Checks-Effects-Interactions (CEI) pattern or use Reentrancy guard modifiers provided by the open-zeppelin library.

```
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 before transferring funds

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

    emit RaffleRefunded(playerAddress);
}
```

Note: It's crucial to update the state before transferring funds to avoid potential re-entrancy vulnerabilities.

Likelihood & Impact:

- Impact : HIGH
- Likelihood : HIGH
- Severity : HIGH

[H-2] Potential Overflow Vulnerability in `PuppyRaffle.sol::updateTotalFees`

Description: There is a potential overflow vulnerability in the `totalFees` calculation. The addition of `fee` to `totalFees` is performed without proper overflow checks. Additionally, unsafe casting with `uint64(fee)` could result in unexpected behavior if the value of `fee` exceeds the maximum representable value of uint64.

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

Impact: The vulnerability may lead to an overflow, causing unexpected behavior and potentially allowing attackers to manipulate the totalFees value.

Recommended Mitigation: Use the latest version of Solidity to benefit from built-in overflow checks. Consider using a larger integer type for totalFees if necessary. Implement proper overflow checks before performing arithmetic operations.

Likelihood & Impact:

- Likelihood: HIGH
- Impact: HIGH
- Severity: HIGH

[M-1] A potential Denial of Service (DoS) attack in `PuppyRaffle.sol::enterRaffle` function while performing an unbounded loop to check for duplicates.

Description: Malicious actors could deliberately exploit the vulnerability to overwhelm the system, causing delays or failures in the raffle entry process. Here's how:

```
 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++) {
         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);
 }
```

While we loop through all the addresses, that is an unbounded loop and hence,as the size of players array increases, the gas costs for the transaction also increases. Therefore, it may lead to excessively high gas costs for the users to enter the raffle in future which will be unfair to them.

Impact: Users attempting to enter the raffle may face disproportionately high gas fees, rendering the process economically impractical.

Proof of Concept: The below test case shows how the gas costs for the transaction increases with the increase in number of players.

```
   function testDOS() public {
     uint gasBef = gasleft();
     address[] memory player = new address[](1);
     player[0]=playerOne;
     puppyRaffle.enterRaffle{value : entranceFee}(player);
     uint gasAft = gasleft();
// gasUsed : gas cost to enter for the first player
     uint gasUsed = gasBef - gasAft;
```

```
        console.log("gas used while adding first member: ",gasUsed);
// total players = 1
        address[] memory players = new address[](48);
        for(uint i=0;i<48;i++){
            players[i] = address(i+2);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
// total players = 49
        uint gas2bef = gasleft();
        address[] memory playernew = new address[](1);
        playernew[0]=address(50);
        puppyRaffle.enterRaffle{value: entranceFee * playernew.length}(playernew);
        uint gas2aft = gasleft();
//gasused2 : gas cost to enter for the 50th player
        uint gasused2 = gas2bef-gas2aft;
        console.log("Gas used while adding 50th member:",gasused2);
        assertTrue(gasused2 > gasUsed);
    }
```

Here is the log output:

```
Gas used while adding first member: 63317 Gas used
while adding 50th member: 1013528
```

Recommended Mitigation: Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
  mapping (address=>bool) addressAdded;
```

Inside enterRaffle function :

```
 for(uint i=0;i<newPlayers.length;i++){
    require(!addressAdded[newPlayers[i]],"Duplicate address added!")
}
```

Likelihood & Impact :

- Impact : MEDIUM
- Likelihood : HIGH
- Severity : MEDIUM

[M-2] Deterministic Winner Index Generation in `PuppyRaffle.sol::selectWinner`

Description: The current method for generating `winnerIndex` using `keccak256` may result in a deterministic number, making it susceptible to manipulation by miners. This deterministic behavior undermines the randomness required for a fair selection of winners.

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

Impact: The lack of true randomness in the winner selection process opens the possibility for miners or external actors to influence the outcome, compromising the fairness of the raffle.

Recommended Mitigation: Consider using more robust methods for obtaining randomness, such as Chainlink VRF (Verifiable Random Function) to ensure a more secure and unbiased winner selection.

Note: Implementing Chainlink VRF requires additional code and integration. Ensure proper research and testing before implementing any solution.

Likelihood & Impact:

- Likelihood: LOW
- Impact: HIGH
- Severity: MEDIUM

[M-3] Vulnerability in Prize Distribution to Smart Contract Winners

Description: There is a potential vulnerability in the prize distribution mechanism to winners. If the winner is a smart contract with a faulty or malicious `receive` function that reverts, the prize pool transfer will fail, and the winner won't receive their funds.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

Impact: If the winner is a smart contract with a misconfigured or malicious receive function, the prize pool transfer will fail, leading to the winner not receiving their funds.

Recommended Mitigation: Implement additional checks to verify whether the winner is an externally-owned account (EOA) rather than a smart contract.

Likelihood & Impact:

- Likelihood: LOW
- Impact: HIGH
- Severity: MEDIUM

[M-4] Potential Mishandling of ETH in `withdrawFees` Function

Description: The `withdrawFees` function is susceptible to potential mishandling of ETH due to the use of an insecure method to check the contract's balance. Checking the contract's own balance for available fees could be exploited through using self-destruct method by any other contract, leading to inaccurate balance checks.

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

Impact: This vulnerability may allow an attacker to force additional funds into the contract through self-destruct attacks, resulting in inaccurate balance checks and potential fund manipulation.

Recommended Mitigation: Implement a more secure approach for tracking and managing available fees, such as maintaining a separate variable to store the actual balance. Additionally, use withdrawal patterns that resist reentrancy attacks to enhance the overall security of the function.

Likelihood & Impact:

- Likelihood: MEDIUM
- Impact: MEDIUM
- Severity: MEDIUM

[L-1] Improvement in getActivePlayerIndex Function Logic

Description: The `getActivePlayerIndex` function currently returns 0 if the provided player address is not found in the players array. This may lead to a misunderstanding, as index 0 could also represent a valid player. This points out that it might seem like the player is not active, causing potential confusion.

Impact: The current implementation may cause confusion, especially when index 0 is returned. It might be interpreted as the player not being active, leading to misinterpretation of the function's result.

Recommended Improvement: Refine the logic to distinguish between the case where the player is not found and the case where the player is found at index 0.

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

    // If a player is not found, return a value that indicates not found, e.g., -1
    return -1;
}
```

Note: Returning a value like -1 when the player is not found allows for a clearer indication that the player is not in the players array.

Likelihood & Impact:

- Impact : LOW
- Likelihood : MEDIUM
- Severity : LOW

[I-1] Use of Floating Pragma Version in `PuppyRaffle.sol`

Description: The pragma statement `pragma solidity ^0.7.6;` specifies a floating version of the Solidity compiler. While this might work for the current version, it is generally not recommended for the following reasons:

Lack of Deterministic Version: A floating pragma allows the compiler to use the latest compatible version, which may introduce unexpected behavior or changes when compiling the contract in the future.

Risk of Breaking Changes: Upgrading to a new compiler version could lead to breaking changes in the contract behavior due to updates in the compiler's features or optimizations.

Impact: The use of a floating pragma version might result in compatibility issues or unexpected changes in contract behavior during future compilations.

Recommended Mitigation: Specify a fixed pragma version to ensure determinism and predictability in your contract's compilation. For example, use pragma solidity 0.7.6; instead of ^0.7.6.

Likelihood & Impact:

- Impact : LOW
- Likelihood : LOW
- Severity : INFORMATIONAL

[I-2] Declaration of raffleDuration as Immutable in `PuppyRaffle.sol`

Description: This suggests that the raffleDuration variable is never changed, and as a best practice (gas efficient), it should be declared as immutable.

```
  uint256 public raffleDuration;
```

Impact: While the current code may function as intended, declaring raffleDuration as immutable provides additional clarity to the readers. If not declared, it may lead to excess gas utilization during the deployment of the contract.

Recommended Mitigation: Consider declaring raffleDuration as immutable to enhance code readability and express the intention that this value remains constant throughout the contract's lifecycle.

Likelihood & Impact:

- Impact : LOW
- Likelihood : LOW
- Severity : INFORMATIONAL

[I-3] Declaration of commonImageUri as Constant in `PuppyRaffle.sol`

Description: This suggests that commonImageUri should be declared as a constant.

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

Impact: While the current code may function as intended, declaring commonImageUri as immutable provides additional clarity to the readers. If not declared, it may lead to excess gas utilization during deployment of the contract

Recommended Mitigation: Consider declaring commonImageUri as constant for improved readability and to convey the immutability of this string value.

Likelihood & Impact:

- Impact : LOW
- Likelihood : LOW
- Severity : INFORMATIONAL

[I-4] Input Validation for feeAddress in Constructor of `PuppyRaffle.sol`

- Description: This suggests adding input validation for the feeAddress parameter in the constructor to check for zero addresses.

Impact: Adding input validation for the feeAddress parameter helps ensure that the contract is initialized with a valid and non-zero Ethereum address, preventing potential issues related to zero or invalid addresses.

Recommended Mitigation: Implement input validation for the feeAddress parameter to check for zero addresses in the constructor.

```
constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration)
ERC721("Puppy Raffle", "PR") {
   // Input validation for feeAddress
   require(_feeAddress != address(0), "PuppyRaffle: Invalid fee address");

   // Rest of the constructor code...
   feeAddress = _feeAddress;

}
```

Likelihood & Impact:

- Impact : LOW
- Likelihood : LOW
- Severity: INFORMATIONAL

[I-5] Optimize loop iteration by Using Memory Variable for `players.length` in `PuppyRaffle.sol`

Description: This suggests optimizing gas usage by storing the length of the players array in a memory variable instead of repeatedly accessing `players.length` in the loop conditions.

Impact: By using a memory variable to store the length of the players array, the gas cost of the loop iteration can be reduced. Accessing storage variables is more expensive than using memory variables.

Recommended Mitigation: Consider storing the length of the players array in a memory variable before entering the loop to optimize gas usage.

```
// Storing the length of players array in a memory variable
uint256 playerLen = players.length;

for (uint256 i = 0; i < playerLen - 1; i++) {
    for (uint256 j = i + 1; j < playerLen; j++) {
        // Loop logic...
    }
}
```

Likelihood & Impact:

- Impact: LOW
- Likelihood: LOW
- Severity: INFORMATIONAL

[I-6] Unused Internal Function: `_isActivePlayer`

Description: The `_isActivePlayer` function is defined in the contract but is never used anywhere in the contract code. This results in unnecessary gas consumption during contract execution, making it a potential waste of resources.

Impact: The presence of an unused function incurs unnecessary gas costs during contract deployment and execution, which may impact the overall efficiency of the contract.

Recommended Mitigation: Remove or comment out the unused _isActivePlayer function to eliminate the gas overhead associated with its deployment and execution. Note: Regularly review and clean up unused or redundant code to optimize gas usage.

Likelihood & Impact:

- Likelihood: Low
- Impact: Low
- Severity: Low