PuppyRaffle Audit Report

Version 1.0

Nathaniel Yeboah

June 21, 2024

PuppyRaffle Audit Report

Nathaniel Yeboah

June 21, 2024

Prepared by: Nathaniel Yeboah Auditor: - Nathaniel Yeboah

Table of Contents

Protocol Summary

This project allows users to enter a reaffle to win a cute dog NFT. The protocol should do the folllowing: 1. player enters the raffle by calling the payable `enterRaffle` function . total entrance fee is based on the the basic entrance fee * the number of participants. 2. Players can choose to get a refund by calling the `refund` function, which sends the entrance fee back to the player. 3. A winner is selected after X amount of time and is minted a puppy. The random selection is based on the hash of the block

difficulty annd msg.sender address 4. 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. 5. The owner can set the address of the feeAddress .

Disclaimer

The Security Researcher 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

** The findings described in this report are based on the code at the following commit hash: **

```
Commit hash: e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

** The audit report is based on the following files: **

```
src/PuppyRaffle.sol
```

Roles

- Players: Can enter the raffle and get a refund and stand a chance for winner the raffle
- Owner: Can set the feeAddress and withdraw fees and deployer the raffle contract

Executive Summary

- We found some major vulnerabilities in the code base and were in constant communication with the developers to understand logic, code design and possible attack vector in the code
- We spend about 10 days with 1 auditor using tools such as foundry, slither , aderyn

Issues found

Severity	Number of issues found
High	5
Medium	4
Low	2
Info	5
Gas	3
Total	19

Findings

High

[H-1] `PuppyRaffle::refund` function sends out ether before updating the user balance causing a possible ReEntrancy attack

Description: `PuppyRaffle::refund` function sends out ether before updating the user balance causing a possible ReEntrancy attack. An attacker can reenter this function multiples times to receive ether till the contract balance is 0. Also there is no reentrancy guard in the `PuppyRaffle::refund` function to prevent it.

Impact: An attacker can drain all the funds in the raffle . The attacker steals all the funds deposited by all users collapsing the protocol.

Proof of Concept: This can be done with following

1. The attacker creates a contract `AttackerReentrancy` which has a function called `attack` .This `attack` function deposit ether into the `PuppyRaffle` contract and then calls for refund at the same time.
2. During refund the attacker calls `PuppyRaffle::refund` function which sends out ether to the attacker contract
3. The attacker has a `AttackerReentrancy::receive` function which recieves ether from the `PuppyRaffle` contract and then calls `PuppyRaffle::refund` function again.

This has be showed in the test code in `PuppyRaffleTest::testAuditPoC_Reentrancy` function

```
1   Balance of the PuppyRaffle before attack:  4000000000000000000
2   Balance of the PuppyRaffle after attack:  0
3   Balance of the attackContract after attack:  5000000000000000000
```

Recommended Mitigation: here are a few recommendations 1. Consider adding Openzeppelin Reentancy guard for the `PuppyRaffle` contract. Then added a nonRentrant modifier to the `PuppyRaffle::refund` function. 2. Consider updating the internal state (balances) before an external calls .This follows the Checks,Effects and Interactins(CEI) pattern. This has been shown below

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

[H-2] `PuppyRaffle::selectWinner` function has a precision loss when calculating the total fees causing an inabilitiy to collect fees after the winner is selected

Description: `PuppyRaffle::selectWinner` function has a precision loss when it calculates the fee by type casting the total fees from uint256 to uint64

```
1    uint256 fee = (totalAmountCollected * 20) / 100;
2 @>  totalFees = totalFees + uint64(fee);
```

In solidity version < 0.8.0 , there is the code above can cause an overflow leading to a reduction in the amount `PuppyRaffle::totalFees` of `PuppyRaffle` contract which will be redeem by the owner.

Impact: The actual fees of the `PuppyRaffle` contract will be reduced which can cause an inabilitiy of the owner to withdraw fees after the winner is selected.

Proof of Concept: This max uint of uint64 is `2*64 -1`, afterwards solidity version < 0.8.0 overflow and start count from zero. This can occur when the the fees collected are greater than `2^64 -1` which is approximately `18.4e**18` or `18.4 ether` . So if the protocol has a fees above 18.4 ethers the fees overflow starting the `totalFees` value from zero.

Consider the code in the test code `PuppyRaffleTest::testAuditPoC_OverFlow` function We know `totalFees` is supposed to increase when the number of player increases.

1. you can set the number of players to 170, 180 ,184, 185 ,186 in the `testAuditPoC_OverFlow` function for each player number

```
1 function testAuditPoC_OverFlow() public {
2     /// code above
3     @>        uint256 numberOfNewPlayers = 186;
4     /// code below
```

2. Observe the logs of by running

```
1 forge test --mt testAuditPoC_OverFlow -vv
```

```
1  Total fees after selecting winner with  170  players is
      15553255926290448384
2
3  Total fees after selecting winner with  184  players is
      18353255926290448384
4
5  Total fees after selecting winner with  185  players is
      106511852580896768
6  Total fees after selecting winner with  186  players is
      306511852580896768
```

3. In the logs above it will be observed that the `totalFees` is for 185 players is less than that of 184 players

Recommended Mitigation: 1. use sathMath or solidity version >0.8.0 2. change the uint64 to uint256 in the code below. 3. remove the uint64 type casting in the code.

```
-    uint64 public totalFees = 0;
+    uint256 public totalFees = 0;

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;       // fee in already uint256
```

[H-3] Winner cannot not receive funds after raffle when there have refunds in a raffle round

Description: Winner cannot not receive funds after raffle when the have refunds in a raffle round due the code logic of the `PuppyRaffle` contract.

1. The `PupppyRaffle::refund` function logic does not reduce the length of the `PuppyRaffle::players` array but replaces it with the `zero address`.

```
 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);
    }
```

2. The `PuppyRaffle::selectWinner` function logic pays the winner of the raffle based on the length of the `PuppyRaffle::players` array to determine the `PuppyRaffle::totalAmountCollected` variable.

```

function selectWinner() public {
    ***Code here***
@>    uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;

    ***Code here***
```

3. In case, there have been refunds , the array length will not be reduced to match the total amount of Eth of the remaining players in the raffle.
4. This would prevent the `PuppyRaffle::selectWinner` from calculating the prize pool correct and causing a revert in the `PuppyRaffle::selectWinner` function.

Impact: The winner cannot be selected and funds would be locked in the `PuppyRaffle` contract

Proof of Concept: Consider the following .

1. Say 6 players entered the raffle.
2. The fifth player and sixth player each calls for the `PuppyRaffle::refund` function and is successfully refunded.
3. Now the `PuppyRaffle::players` array length should be four instead of six.
4. Yet the logic of the `PuppyRaffle::refund` function does not reduce the length of the `PuppyRaffle::players` array.
5. In case of refunds , the `PuppyRaffle::selectWinner` function logic is expected to pay the winner of the raffle based on the length of the `PuppyRaffle::players` array(which is actually four instead instead) to determine the `PuppyRaffle::totalAmountCollected` variable.
6. The contract may not have enough funds to pay the winner of the raffle causing a revert.

Copy and Paste this code here into the test file "'puppyRaffleTest.t.sol""

```
function test_AuditCannotSelectWinnerAfterSomeRefunds() public
    playersEntered {
        // four players have already entered the raffle , check the
            modifier in this function
        //  two  new players enter and two players will be refunded
        address playerFive = makeAddr("5");
        address playerSix = makeAddr("6");
        address[] memory newplayers = new address[](2);
        newplayers[0] = playerFive;
        newplayers[1] = playerSix;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(newplayers);
        console.log("Balance of the contract after entering          :"
            , address(puppyRaffle).balance);

        vm.startPrank(playerSix);
        puppyRaffle.refund(puppyRaffle.getActivePlayerIndex(playerSix))
            ;
        console.log("Balance of the contract after refunding player 6:"
            , address(puppyRaffle).balance);
        vm.stopPrank();

        vm.startPrank(playerFive);
        puppyRaffle.refund(puppyRaffle.getActivePlayerIndex(playerFive)
            );
        console.log("Balance of the contract after refunding player 5:"
            , address(puppyRaffle).balance);
        vm.stopPrank();

        // uint256 balanceBefore = address(playerFour).balance;
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        console.log("Number of players in the raffle after some refunds
            ", puppyRaffle.getNumberOfPlayers());
```

```
27          console.log("Expected number of players after refunds", 4);
28
29          // uint256 expectedPayout = ((entranceFee * 4) * 80 / 100);
30          vm.expectRevert();
31          puppyRaffle.selectWinner();
32          // assertEq(address(playerFour).balance, balanceBefore +
                expectedPayout);
33      }
```

Recommended Mitigation: The `PuppyRaffle::refund` function should reduce the length of the `PuppyRaffle::players` array to match the total amount of Eth of the remaining players in the raffle.

[H-4] Weaker Random Number Generator(WRNG) in `PuppyRaffle::selectWinner` causing a winner to a predictable

Description: Blockchain are deterministic systems and generating a random number is not random since the random number generated on chain can be determined.

Impact: Miners and Malicious attackers can predict the winner to suit their needs.

Proof of Concept: 1. This code is in the test `PuppyRaffleTest::testSelectWinner` to select the winner.
2. Whenever is code below is run, it will always select the playerFour as the winner all the time . This is not a random .

```
1       function testSelectWinner() public playersEntered {
2           vm.warp(block.timestamp + duration + 1);
3           vm.roll(block.number + 1);
4
5           puppyRaffle.selectWinner();
6           assertEq(puppyRaffle.previousWinner(), playerFour);
7       }
```

Recommended Mitigation: Use an orcle such as ChainLink VRF . The ChainLink Docs are available here.

[H-5] Weak Random Number Generator(WRNG) in `puppyRaffle::selectWinner::rarity`

Description: A keccak hash of msg.sender and block.difficulty doesnot generate a random number

Impact: this is predicatable as the previous finding in H-4 . The attacker can predict the winner to suit their needs.

Recommended Mitigation: Should use ChainLink VRF.The ChainLink Docs are available here.

Medium

[M-1] `PuppyRaffe::enterRaffle` function has unbounded loop leading to a potential Denial of Service(DoS) attack , increasing the gas costs for future entrace

Description: `PuppyRaffe::enterRaffle` function has a for loop that is unbounded, an attacker can enter the raffle with a large number of array causing a spike in gas for subsequent calls.

Code Snippet in `PuppyRaffe::enterRaffle`

```
1
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          }
7          emit RaffleEnter(newPlayers);
8      }
```

Impact: The gast 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 enterants in the queue. An attacker can enter the raffle with a large number of players at the start which will cause a spike in gas costs , preventing others users from using garanting a win

Proof of Concept: It is possible to enter the raffle with a large number of players if the attacker is well funded. Moreover legitimate users can overwhelm cause a DDoS when the array to loop become very large

The test code in `PuppyRaffleTest::testAuditPoC_DoS` shows the raise in gas Cost after more large numbers of players have entered the contract

Code Snippet in `PuppyRaffleTest::testAuditPoC_DoS`

```
1  function testAuditPoC_DoS() public {
2          uint256 numberOfNewPlayers = 100;
3          address[] memory newPlayers = new address[](numberOfNewPlayers)
               ;
4          for (uint256 i = 0; i < numberOfNewPlayers; i++) {
5              newPlayers[i] = address(i);
6          }
7          uint256 gasStart = gasleft();
8
9          puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length
               }(newPlayers);
10        uint256 gasEnd = gasleft();
11
12 @>>          uint256 gasUsedFirst = gasStart - gasEnd;
```

```
13          console.log("GasUsed by player 1 after entering raffle is: ",
                gasUsedFirst);
14
15          // a second player enters the raffle
16
17          address[] memory newPlayers2 = new address[](numberOfNewPlayers
                );
18          for (uint256 i = 0; i < numberOfNewPlayers; i++) {
19              newPlayers2[i] = address(i + numberOfNewPlayers);
20          }
21          uint256 gasStart2 = gasleft();
22
23          puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length
                }(newPlayers2);
24          uint256 gasEnd2 = gasleft();
25
26  @>>      uint256 gasUsedSeecond = gasStart2 - gasEnd2;
27          console.log("GasUsed by player 2 after entering raffle is: ",
                gasUsedSeecond);
28          assert(gasUsedSeecond > gasUsedFirst);
29      }
```

Recommended Mitigation: Therea are few recommendations.

1. Consider allowing the duplicates.Users can make new wallet addresses anyways, so a check doesnt prevent the same person from entering multiiple times in the raffle.
2. Use a mapping instead of a list.

```
1 mapping (address Player => bool hasEntered) PlayersEntered;
2 for (uint256 i = 0; i < players.length - 1; i++) {
3     if (PlayersEntered[players[i]]) {revert("PuppyRaffle: Duplicate
          player");}
4 }
```

[M-2] `PuppyRaffle::withdraw` function should be callable by anyone which can lead manipulation by others

Description: `PuppyRaffle::withdrawal` function should be callable by anyone which is lead manipulation by others , for instance the fee address set by the owner might unable to accept eth/is a wrong address . An attacker will call this function to send out the fees

Impact: Anyone can call the `PuppyRaffle::withdraw` function to send out the fees which is only intended by the owner

Recommended Mitigation: Add the onlyOwner modifier to the `PuppyRaffle::withdraw` function

```
1 -     function withdrawFees() external {
2 +     function withdrawFees() external onlyOwner {
```

[M-3] Mishandling of Eth in `PuppyRaffle::withdraw` function causing the owner unable to withdraw fees after raffle

Description: The logic inside the `PuppyRaffle::withdraw` function strictly checks that the totalFees should be always equal to the balance of the contract.

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

With this assertion , a malicious user can forcily push eth into the contract using self destruct , this will raise the balance of the contract to be greater than the fees , breaking the === assertion.

Impact: The owner will be unable to withdraw the fees even after the raffle.

Recommended Mitigation: Remove the check from the code

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

[M-4] Smart Contracts who might be winner of the raffle may not have a `fallback` or `receive` function to receive eth causing the `PuppyRaffle::selectWinner` function to revert leading to inability to reset the players and start a new raffle

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the players and starting a new raffle. Howerver , if the winner of the raffle is a smart contract and rejects eth , the `PuppyRaffle::selectWinner` function will revert.Hence the raffle will not be able to restart

Impact: Smart Contracts winners cannot receive funds , and `PuppyRaffle` contract cannot start a new raffle

Proof of Concept:

1. if only smart contracts wallet enters the raffle without a `fallback` or `receive` function.
2. the lottery ends .
3. the `PuppyRaffle::selectWinner` function would not work as expected and the `PuppyRaffle` contract would not be able to start a new raffle.

Recommended Mitigation: 1. Do not allow smart contracts to enter the raffle(not recommended). 2. Create a mapping of address to payout , so that winners can pull their funds out themselves with a

new `claimPrize` function , putting the ownes on the wineer to claim their prize (recommended). 3. this is a pull over push preferences .

Low

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

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

```
    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 playeer 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. The first player might think they are not active since the `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players according to the natspec

Recommended Mitigation: Should change the natspec documentation and set players that are not active to -1 Also should change the `PuppyRaffle::getActivePlayerIndex` to return -1 if the player is not active

```
-    function getActivePlayerIndex(address player) external view returns (uint256) {
+    function getActivePlayerIndex(address player) external view returns (int256) {

        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
-        return 0;
+        return -1;
    }
```

[L-2]: Solidity pragma should be specific, not wide

Description: 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;`

Code here

- Found in src/PuppyRaffle.sol Line: 4

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

Impact: Some specific version of solidity are not stable and would led to bugs in the code

Recommended Mitigation: Use a specfic and stable version of solidity preferrably version 0.8.0 . More Info on Slither Documentation

Informational

[I-1]: 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: 82

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 235

Recommended Mitigation: Add checks for `address(0)` when setting the `feeAddress` state variable

[I-2] The `PuppyRaffle::enterRaffle` and `PuppyRaffle::refund` function should marked as external and not public

Description: `PuppyRaffle::enterRaffle` and `PuppyRaffle::refund` functions were not used internally and should be marked as `external`

Recommended Mitigation: for `PuppyRaffle::enterRaffle` function,

```
1  - function enterRaffle(address[] memory newPlayers) public payable {
2  + function enterRaffle(address[] memory newPlayers) external payable {
```

for `PuppyRaffle::refund` function,

```
- function refund(uint256 playerIndex) public {
+ function refund(uint256 playerIndex) external {
```

[I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not the best practice

Recommended Mitigation:

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

[I-4] Should not Magic Numbers in `PuppyRaffle` contract be set to contacts

Description: In `PuppyRaffle::selectWinner` function, magic numbers should not be used.

```
function selectWinner() external {
        ** Code here **
@>        uint256 prizePool = (totalAmountCollected * 80) / 100;
@>       uint256 fee = (totalAmountCollected * 20) / 100;
      *** Code here ***
    }
```

Impact: This will make the code unreadable or difficulty to understand what the numbers means

Recommended Mitigation: set those magic numbers to constants in the contract

```
+ uint256 public constant WINNER_PERCENTAGE = 80;
+ uint256 public constant FEE_PERCENTAGE = 20;
+ uint256 public constant PRECISION_PERCENTAGE = 100;


function selectWinner() external {
        ** Code here **
-       uint256 prizePool = (totalAmountCollected * 80) / 100;
+       uint256 prizePool = (totalAmountCollected * WINNER_PERCENTAGE)
    / PRECISION_PERCENTAGE;
-      uint256 fee = (totalAmountCollected * 20) / 100;
+      uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
    PRECISION_PERCENTAGE;
      *** Code here ***
    }
```

[I-5] Unused `PuppyRaffle::_isActivePlayer` function making the codebase unreadable

Description: The internal `PuppyRaffle::_isActivePlayer` function was never called in any other part of the `PuppyRaffle` contract

Recommended Mitigation: Delete the whole function

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

Gas

[G-1] `PuppyRaffle::players` Public Arrays Variable Should be Set to Private Arrary To Save Gas

Description: It consumes gas to set an array variable public.

Proof of Concept:

Recommended Mitigation: You should set the variable to private and use a getter function to access the index of array

```
- address[] public players;
+ address[] private players;

+  function getPlayers(uint256 index) public view returns (address) {
+    players[index];
+  }
```

[G-2] Unchanged State Variables in the should be set to immutable or constant to save gas.

Reading from stroage is much more expensive than reading from a 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`.

Recommended Mitigation:

```
- uint256 public raffleDuration;
+ uint256 public immutable raffleDuration;
```

[G-3] `PuppyRaffle::players.length` in `PuppyRaffle::enterRaffle` function should use cached array length instead of referencing `length` member of the storage array.

Recommended Mitigation: In the case of `PuppyRaffle::enterRaffle` function, it is recommended to use the array length instead of referencing the `length` member of the storage array.

```
 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]);
        }
+   unit256 playersLength = players.length;

-   for (uint256 i = 0; i < players.length - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
+   for (uint256 i = 0; i < playersLength  - 1; i++) {
+           for (uint256 j = i + 1; j < playersLength ; j++) {
              require(players[i] != players[j], "PuppyRaffle:
                  Duplicate player");
            }
        }
}}}
```