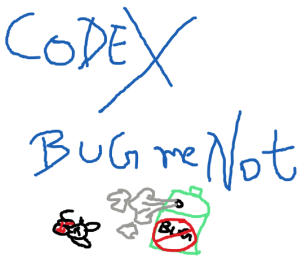

Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

April 20, 2024

Puppy Raffle Audit Report

CodexBugMeNot

April 20, 2024

Prepared by: [CodexBugMeNot] Lead Auditors: - CodexBugMeNot

Table of Contents

  - [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle()` is a potential denial of service(DOS) attack increasing the gas cost for future entrants
  - [M-2] Unsafe Casting in `PuppyRaffle::selectWinner`
  - [M-3] Smart Contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

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

- Gas
  - [G-1]: Unchanges state Variables should be declared constant of immutable
  - [G-2]: Storage Variables in a Loop should be cached

- Informational/Non-Crit
  - [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  - [I-2]: Using an outdated version of Solidity is not recommended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]: `PupyRaffle::selectWinner` doesn't follow CEI, which is not a best practice
  - [I-5]: Use of "magic" numbers is discouraged
  - [I-6]: State Changes are Missing Events
  - [I-7]: `PuppyRaffle::_isActivePlayer` is never used and should be removed

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 CodexBugMeNot 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

```
./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 I enjoyed a lot auditing Puppy Raffle protocol. ## Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	2
Info	7
Total	17

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 raffle contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after that external call do we update the `PuppyRaffle::players` array.

```
1
2   function refund(uint256 playerIndex) public {
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   @>      payable(msg.sender).sendValue(entranceFee);
8   @>      players[playerIndex] = address(0);
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 till the contract balance is drained completely.

Impact: All fees paid by the raffle entrants can be drained by a malicious player.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a cpntact with a `fallback()` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Sttacker calls `PuppyRaffle::refund` fropm their attack contract draining the contract balance

Code

Place the following piece of code into `PuppyRaffle.t.sol`.

```
1
2  function testRefundReentrancy() public {
3          Attacker attack = new Attacker(puppyRaffle);
4          address[] memory players = new address[](5);
5          players[0] = address(attack);
6          players[1] = playerOne;
7          players[2] = playerTwo;
8          players[3] = playerThree;
9          players[4] = playerFour;
10         puppyRaffle.enterRaffle{value: entranceFee * 5}(players);
11
12         uint256 balanceBefore = address(puppyRaffle).balance;
13
14         uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
               address(attack));
15
16         vm.prank(address(attack));
17         //puppyRaffle.refund(indexOfPlayer);
18         attack.attack(indexOfPlayer);
19
20         uint256 balanceAfter = address(puppyRaffle).balance;
21
22         console.log("PuppyRaffle Balance Before Attack:", balanceBefore
               );
23         console.log("PuppyRaffle Balance After Attack:", balanceAfter);
24         console.log("Attacker Balance:", address(attack).balance);
25
26         assert(address(attack).balance > 1 ether);
27         assert(balanceAfter < balanceBefore - 1 ether);
28         assertEq(address(attack).balance, balanceBefore);
29     }
```

And also this contract.

```
1
2  contract Attacker {
3      PuppyRaffle victim;
4      uint256 index;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          victim = _puppyRaffle;
8      }
9
10     function attack(uint256 _index) public payable {
11         index = _index;
12         victim.refund(index);
13     }
14
15     receive() external payable {
16         if (address(victim).balance > 0) {
17             victim.refund(index);
18         }
19     }
20 }
```

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 as well.

```
1
2     function refund(uint256 playerIndex) public {
3         address playerAddress = players[playerIndex];
4         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
6
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    }
```

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

Note: This additionally means users could front-run and call `refund` if they see they are not the winner

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

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

Using on-chain values as a randomness send 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 subject to integer overflow.

```
1
2 uint64 myVar = type(uint64).max;
3 // myVar is now 18446744073709551615
4 myVar = myVar + 1;
5 // myVar here will become 0
```

Impact: In `PuppyRaffle::selectWinner`,`totalFees` are accumulated for `feeAddress` to collect later in `PuppyRaffle::withdrawFees` , However if the `totalFees` variable overflows the feeAddress may not collect the correct amount of fees, leaving sees permanently stuck in the contract.

Proof of Concept:

1. when more than 93 players's fee is collected in total fees it overflows

2. You will not be ble to withdraw fees from `PuppyRaffle:withdrawFees` due to this line

```
1
2 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 and withdraw the fees , this is clearly not the intended design for the protocol.At some point , there will be too much `balance` in the contract that the above `require` will be impossible to hit.

PoC

```
1
2 function testTotalFeesOverFlows() public {
3         uint64 length = 93;
4
5         address[] memory players = new address[](length);
6         for (uint256 i = 0; i < length; i++) {
7             players[i] = address(i);
8         }
9         puppyRaffle.enterRaffle{value: entranceFee * length}(players);
10
11        uint256 BeforeOverFlow = puppyRaffle.totalFees();
12        vm.warp(duration + 1);
13        // vm.roll(block.number + 1);
14
15        puppyRaffle.selectWinner();
16
17        uint256 AfterOverFlow = puppyRaffle.totalFees();
18        console.log("Before OverFlow:", BeforeOverFlow);
19        console.log("After OverFlow:", AfterOverFlow);
20
21        uint256 FeesWithoutOverFlow = uint256(18400000000000000000);
22
23        assert(AfterOverFlow < FeesWithoutOverFlow);
24    }
```

Recommended Mitigation: There are a 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'll still have a hard time eith the `uint64` type if too much fees is collected.
3. Remove the balance check from `PuppyRaffle::withdraeFees`

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

There are more attack vectors with the above `require` so we recommend to remove it regardless.

[H-4] In `PuppyRaffle::refund` not deleting the player index from the `players` array makes `address(0)` a possible contender for select winner

Description: In `PuppyRaffle::refund` whenever a player requests a refund instead of deleting his entry from the `players` array the code changes the player address to `address(0)` making zero-address a possible contender for select winner.

Impact: 1. If multiple players request a refund then there will be multiple `address(0)` contenders for select winner, which breaks the `duplicate players are not allowed` invariant of the contract 2. If there are multiple `address(0)` contending then there is an increased chance that zero address will be selected as the winner and more Chances for a DOS(denial of service) in `PuppyRaffle::selectWinner` since `address(0)` doesn't have a `fallback` function and can never receive the prizeamount. 3. A Malicious user can enter the raffle and request refund multiple times to perform this DOS attack permanently locking the contract.

Proof of Concept:

PoC

This is a case where a player refunds , making zero address a winner and the selectWinner function always reverts.

```
1
2  function testNotDeletingPlayersAfterRefundMakesAddressZeroWinner()
       public {
3          uint256 length = 4;
4          address[] memory players = new address[](length);
5          players[0] = address(5);
6          for (uint256 i = 1; i < length; i++) {
7              players[i] = address(i);
8          }
9          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
10
11         for (uint256 i = 1; i < length; i++) {
12             vm.prank(players[i]);
13             puppyRaffle.refund(i);
14         }
15
16         vm.warp(duration + 1);
17         // vm.roll(block.number + 1);
18
19         vm.expectRevert();
20         puppyRaffle.selectWinner();
21     }
```

Recommended Mitigation: Instead of changing the refunded player index to `address(0)` swap the player to with the last index and pop that player from the array which helps to keep the array length in

check to call `selectWinner`

```
1
2     function refund(uint256 playerIndex) public {
3         address playerAddress = players[playerIndex];
4         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
5         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
6
7
8         payable(msg.sender).sendValue(entranceFee);
9
10 -       players[playerIndex] = address(0);
11 +       address temp;
12 +       temp = players[playerIndex];
13 +       players[playerIndex] = players[length - 1];
14 +       players[length - 1] = temp;
15 +       players.pop();
16        emit RaffleRefunded(playerAddress);
17    }
```

Medium

[M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle()` is a potential denial of service(DOS) attack increasing the 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 player will have to make.This mwans the gas cost for the players who enter right when the raffle starts will be dramaticall lower than those who enter later . Every address in the `players` array is an additional check the loop will have to make.

```
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          }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle . Discouraging later users from entering and causing the rush at the start of a raffle to be one of the first entrants in the queue.

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

Proof of Concept: For every added player in the raffle the gas cost of the next to enter the raffle increases

Player 1 Gas : 63317 – Player 1 Gas is ignored since it always consumes more gas (as it initiates the storage variable `players`)

Player 2 Gas : 35981 Player 3 Gas : 37849 Player 4 Gas : 40506

It is clear that gas used increases exponentially as the number of players increase

PoC

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

```

  function testDosInEnterRaffle() public {
       uint256 startGas1 = gasleft();
       address[] memory players = new address[](1);
       players[0] = playerOne;
       puppyRaffle.enterRaffle{value: entranceFee}(players);
       uint256 playerOneGas = startGas1 - gasleft();

       uint256 startGas2 = gasleft();
       address[] memory players1 = new address[](1);
       players1[0] = playerTwo;
       puppyRaffle.enterRaffle{value: entranceFee * 1}(players1);
       uint256 playerTwoGas = startGas2 - gasleft();

       uint256 startGas3 = gasleft();
       address[] memory players2 = new address[](1);
       players2[0] = playerThree;
       puppyRaffle.enterRaffle{value: entranceFee}(players2);
       uint256 playerThreeGas = startGas3 - gasleft();

       uint256 startGas4 = gasleft();
       address[] memory players3 = new address[](1);
       players3[0] = playerFour;
       puppyRaffle.enterRaffle{value: entranceFee}(players3);
       uint256 playerFourGas = startGas4 - gasleft();

       console.log("Player 1 Gas :", playerOneGas);
       console.log("Player 2 Gas :", playerTwoGas);
       console.log("Player 3 Gas :", playerThreeGas);
       console.log("Player 4 Gas :", playerFourGas);

       assert(playerTwoGas < playerThreeGas);
       assert(playerThreeGas < playerFourGas);
```

```
34          assert((playerThreeGas - playerTwoGas) < (playerFourGas -
               playerThreeGas));
35      }
```

Recommended Mitigtion: There are a few recommendations

1. Consider allowing duplicates. users can make new wallet addresses anyways , so a duplicate check doesn't prevent a player entering multiple times, only the same wallet address.
2. Consider using mapping to check for duplicates . This would allow a constant time lookup of whether a user has already entered.
3. Alternatively you could use OpenZeppelin's`EnumerableSet` library

[M-2] Unsafe Casting in `PuppyRaffle::selectWinner`

Description: In solidity versions prior to `0.8.0` if there is an unsafecasting then values may over/under flow .

```

   totalFees = totalFees + uint64(fee);
```

Impact: If the total fee crosses 20e18 the `uint64(fee)` value overflows ,as it is greater than uint64's range

Proof of Concept: An Example of How the `totalfees` overflows

PoC

```

    function testUnsafeCasting() public {
        uint256 fee = 20e18;

        uint64 totalfees = uint64(fee);

        console.log("Total Fees Before OverFlow:", totalfees);

        assert(totalfees < fee);
    }
```

Recommended Mitigation: Unsafe casting should be avoided. make the following changes in the code

```
-     totalFees = totalFees + uint64(fee);
+     totalFees = totalFees + fee;
```

[M-3] Smart Contract wallets raffle winners without a `receive` or a `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 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 would cost a lot due to the duplicate check and the lottery reset could get very challenging. **Impact:** The `PuppyRaffle::selectWinner` function could revert many times making the lottery reset difficult.

Also , 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 wouldn't work , even though the lottery is over.

Recommended Mitigation: There are a few options to mitigate this issue

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses –> payout so that winners could pull their funds themselves with a new `claimPrize` function , putting the responsibility on the winner to claim their prize (Recommended).

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 for non-existant players and for players at inder 0 causing the players at index 0 to 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 theplayer 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
9           return 0;
10      }
```

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

Recommended Mitigation: One 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 for any competition., but a better solution should be to retuan an `int256` where the function return `-1` when the player is not in the array.

Gas

[G-1]: Unchanges state Variables should be declared constant of immutable

Reading from storage is much more expensive than reading from constant or 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 Variables in a Loop should be cached

Everytime you call `players.length` , you read from storage as opposed to memory which is more gas efficient.

```
1  +    uint256 playerLength = players.length;
2  -    for (uint256 i = 0; i < players.length - 1; i++) {
3  +     for (uint256 i = 0; i < playerLength - 1; i++) {
4  -            for (uint256 j = i + 1; j < players.length; j++) {
5  +             for (uint256 j = i + 1; j < playerLength; j++) {
6                   require(players[i] != players[j], "PuppyRaffle:
                        Duplicate player");
7               }
8           }
```

Informational/Non-Crit

[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

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

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

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

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] (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more info

[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: 62

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 150

```
1         previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 168

```
1         feeAddress = newFeeAddress;
```

[I-4]: `PupyRaffle::selectWinner` doesn't follow CEI, which is not a best practice

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

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

[I-5]: Use of "magic" numbers is discouraged

It would be confusing to see the number literals ina code base and it would be much more readable if the numbers are given a name.

Examples:

```
1
2     uint256 prizePool = (totalAmountCollected * 80) / 100;
3     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 are Missing Events

The functions `PuppyRaffle::selectWinner` and `PuppyRaffle::withdrawFees` do not emit events for state changes which should be emitted.

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

`PuppyRaffle::_isActivePlayer` is never used anywhere in the protocol hence it can be safely removed to save some deployment costs.