audit-report/report-formated.md

---

title: PuppyRaffle Audit Report author: Rao Himanshu Yadav date: June 6,2024 header-includes:

- \usepackage{titling}
- \usepackage{graphicx}

\begin{titlepage} \centering \begin{figure}[h] \centering \includegraphics[width=0.5\textwidth]{logo.pdf} \end{figure} \vspace*{2cm} {\Huge\bfseries PuppyRaffle Audit Report\par} \vspace{1cm} {\Large Version 1.0\par} \vspace{2cm} {\Large\itshape Himanshu\par} \vfill {\large \today\par} \end{titlepage}

\maketitle

Prepared by: [Himanshu] Lead Auditors:

- Rao Himanshu Yadav

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.

Disclaimer

The YOUR_NAME_HERE 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: 22bbbb2c47f3f2b78c1b134590baf41383fd354f
- In Scope:

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

I loved Auditing this codebase, Himanshu is wizard and happy to complete this practice audit report .

Issues found

SEVERITY	NUMBER OF ISSUES FOUND
High	2
Medium	3
Low	1
Info	7
Gas	2
Total	15

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 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), "PuppyRaffle: Player already refunded, or is not active");


@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who entered the raffle could have a `fallback`/`recieve` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could countinue 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 enter 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 balance.

Proof of Code

▶ Code

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.

```
    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::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 find number. A predictable number is not a good number. MAlicious users can manipulate these values or know them ahead of line of time to choose the winner of the raffle themselves.

Note: This additionally means user 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 the gas war as to who wins the raffles.

Proof of Concept:

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 prevrandao.`block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in the address being used to generate the winner!
3. User can revert `selectWinner` transaction if the 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.

Medium

[M-1] Integer overflow of `PuppyRaffle::totalFees` lose fees

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

```
uint64 myVar = type(myVar).max
//18446744073709551615
myVar = myVar + 1
// myVar will 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 the fees, leaving fees permanently stuck in the contract

Proof of Concept:

1. We conclude a raffle of 4 players
2. We have 89 players entered in new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
//aka
totalFees = 800000000000000000 + 1780000000000000000
// and this will overflow!
totalFees = 153255926290448384
```

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 indeed design of the protocol.

▶ Code

Recommended Mitigation:

[M-2] Looping through players to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack,incrementing gas costs for future entrants.

IMPACT: MEDIUM LIKELIHOOD: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates.However, the longer the to make. This means the gas costs for players who enter right when the raffle stats 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.

```
// This is denial of service (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 greatly increase as more players enter the raffle.Discouraginglater 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::entrants` 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 a such: -1st 100 players: ~6252048 gas -2nd 100 players: ~18068138 gas

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

▶ PoC

Recommended Mitigation: There are few recommandations.

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 mapping to check for duplicates.This would allow constant tome lookup of whether a user has already entered.

[M-3] Smart Contract wallet 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.

User 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 winner 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 migrate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amounts 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::getActivePlayersIndex` return 0 for non-existent players and for players at index 0,causing 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.

```
    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 entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. Usr entered the raffle, they are the first entrant. 2.`PuppyRaffle::getActivePlayerIndex` return 0.
2. 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 any7 competition, but a better solution might be return an `int256` where the function returns -1 if the player is not active.

Gas

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

Reading from storage is much 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.

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

```
+       uint256 playerLength = players.length();
-       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");
            }
        }
```

[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

Informational

[I-2] Using an outdated version of Solidity is not recommanded.

Please use latest version of solidity.That include new function tyhat gives more impact on code base.

solc frequently release new compiler version. 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 recommendation 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 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

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

It's 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 "magic" numbers is disscouraged

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

Example:

```
        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;
```

[I-6] Sate changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never be used and should be removed