Prepared by: [DanielCawley]

High

[H-1] Denial of service attack from looping through players array

Description: With each progressing entrance of more users to the array ‘`PuppyRaffle::players`, gas costs become progressively more expensive for users, providing earlier users a financial advantage in the game due to their apparent lower entrance fee.

Impact: Increasing costs **Proof of Concept:** Here suggests how the gas costs significantly rise following new additions of people due to the nature of the unbounded for loop. Unbounded for loop:

```
for (uint256 i = 0; i < players.length - 1; i++) { //not sure about this one
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```

Test case:

```
function testDOSAttackVector() public {


         address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = (address(i));
        }


        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirstHundredPlayers = gasStart-gasEnd;

        console.log("gas cost of first hundred players:", gasUsedFirstHundredPlayers );

     address[] memory players2 = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players2[i] = (address(i + 100));
        }

```

```
23
24          uint256 gasStart2 = gasleft();
25          puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
                players2);
26          uint256 gasEnd2 = gasleft();
27
28          uint256 gasUsedFirstHundredPlayers2 = gasStart2-gasEnd2;
29
30          require(gasUsedFirstHundredPlayers2 >
                gasUsedFirstHundredPlayers);
31      }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, soa duplicate check doens't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow for constnat time loopup of whether a user has already entered.

[H-2] Reentrancy attack in `PuppyRaffle::Refund` allows entrant to drain raffle balance

IMPACT: HIGH LIKELIHOOD: HIGH

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

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

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6
7          payable(msg.sender).sendValue(entranceFee);
8
9          players[playerIndex] = address(0);
10         emit RaffleRefunded(playerAddress);
11     }
```

A player who has just entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle:refund` function again and claim another refund. They could continue the cycle to fully drain the contract balance.

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

Proof of concept:

1. User enters raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function from their attack contract's `fallback`/`receive` function, fully draining the contract balance.

Recommended Mitigation Follow the CEI pattern

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

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

```
uint64 myVar = type(uint64).max
myVar = myVar + 1
// 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` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

```
    function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

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

```
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended Mitigation:

1. Use a newer version of solidity
2. Use openzeppelin `SafeMath` library
3. Use a uint256
4. Remove the balance check from `PuppyRaffle::withdrawFees`

Low secerity

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

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to hte natspec, it will also return 0 if the player is not in the array.

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

Impact: A player at index 0 may incorrectly think that 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
3. User tihnks 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.

Gas

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

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

Instances:

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

[G-2] Storage variables in a loop, should be cached to avoid wasting excess gas

Everytime you read players.length, you read from storage, as opposed to reading from memory which is more gas efficient.

```
- for (uint256 i = 0; i < players.length - 1; i++) {
+ uint256 playersLength = players.length;
+ 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");
            }
        }```
```