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

Description: In the ThunderLoan contract, The `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way it is responsible for 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 impact to this bug.

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

Proof of Concept:

1. Lp deposits.
2. User takes out flashloan.
3. it is now impossible for the Lp to redeem.

▶ Proof of Code

Recommended Mitigation: To resolve this issue, remove the updateExchangeRate call in the deposit function. Since no fee is actually collected during the deposit process, updating the exchange rate here is misleading and results in erroneous protocol behavior.

```
  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);
        // Removed the unnecessary exchange rate update
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[M-1] Oracle Manipulation Reduces Fees (Improper Oracle Integration + Financial Impact)

Description

The ThunderLoan contract relies on an on-chain price oracle derived from the Tswap pool to calculate fees for flash loans. However, the price oracle is manipulable by directly interacting with the Tswap pool, allowing an attacker to distort the asset price during fee calculation. This vulnerability enables the attacker to reduce the fees paid for flash loans, leading to a financial loss for the protocol.

Impact

An attacker can manipulate the oracle to reduce the flash loan fees, as demonstrated in the test case. The manipulation resulted in a reduction of fees from 296147410319118389 to 214167600932190305, indicating a 27.7% fee reduction. Such exploitation could accumulate significant losses over multiple attacks, draining protocol funds and undermining user trust.

Proof of Concept

Initial Setup:

Fund the Tswap pool with 100 WETH and 100 tokenA. Set up ThunderLoan and deposit 1000 tokenA for flash loans. Attack Execution:

Take out an initial flash loan of 50 tokenA and use it to nuke the WETH/tokenA price on Tswap by swapping 50 tokenA for a reduced amount of WETH. This manipulation lowers the value of tokenA relative to WETH, directly affecting the oracle-reported price. Reduced Fees:

Take out another flash loan of 50 tokenA after the manipulation. Due to the reduced oracle price of tokenA, the flash loan fees are significantly lower. Test Logs:

Normal Fees: 296147410319118389 Attack Fees (combined): 214167600932190305

Recommended Mitigation

1. Use a Robust Oracle: Replace the Tswap-based price oracle with a more robust solution, such as Chainlink or a time-weighted average price (TWAP) mechanism, to mitigate short-term price manipulations.
2. Price Validation: Add safeguards to validate the oracle price against external reliable sources or expected price ranges before fee calculations.

3. Flash Loan Fee Calculation: Decouple flash loan fees from manipulable on-chain price oracles. Instead, use an independent mechanism or a fixed fee model.

4. Price Impact Controls: Introduce checks on large swaps within Tswap pools to limit abrupt price changes that can influence dependent oracles.

[H-2] Flash Loan Repayment Vulnerability: Deposit Function Misuse (Incorrect Accounting + Fund Theft)

Description: The smart contract DepositOverRepay demonstrates a vulnerability in the ThunderLoan protocol, where a malicious user can exploit the `deposit` functionality to steal funds during a flashloan. The root cause lies in the contract allowing deposits of flashloaned funds instead of requiring immediate repayment. By depositing the borrowed amount along with the fee, the attacker gains additional shares or tokens representing the deposited value. These shares are then redeemed to withdraw more funds than initially borrowed, effectively stealing funds from the protocol.

Impact: The impact of this vulnerability is severe. It could lead to:

1. Incorrect accounting of flash loan repayments
2. Potential theft of funds from the protocol
3. Imbalance in the protocol's asset pools
4. Loss of user funds who have deposited into the protocol

This vulnerability undermines the core functionality and security of the flash loan feature, potentially leading to significant financial losses for the protocol and its users.

Proof of Concept: The vulnerability can be demonstrated through the following steps:

1. A malicious user creates a contract designed to exploit this vulnerability.
2. This contract initiates a flash loan from the ThunderLoan protocol for a specific amount of tokens.
3. When the flash loan is executed, instead of calling the repay function as intended, the malicious contract calls the deposit function with the borrowed amount plus the fee.
4. By using deposit instead of repay, the contract potentially receives asset tokens in return, as if it were making a regular deposit.
5. The malicious contract then immediately redeems these asset tokens.
6. If successful, the malicious contract ends up with more tokens than it initially borrowed plus the fee, effectively stealing funds from the protocol.

Recommended Mitigation: To mitigate this vulnerability, consider implementing the following measures:

Separate the accounting for flash loan repayments and regular deposits. Do not allow the deposit function to be used for flash loan repayments. Implement a specific repayFlashLoan function that must be called to settle flash loans. This function should not mint any asset tokens or affect the user's deposit balance. Add a check in the deposit function to ensure it's not being called during a flash loan transaction.

[H-3] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol.

Description: `ThunderLoan.sol` has two storage variables, in the following order:

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

However, The `ThunderLoanUpgraded.sol` contract has the storage variables in different order: ```solidity uint256 private s_flashLoanFee; uint256 public constant FEE_PRECISION = 1e18;

```
```
```

Due to how solidity works with storage, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision` you cannot adjust the position of the storage variable and removing storage variable for constant variable, breaks the storage locations as well as the logic of the contract.

**Impact:** After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision` and the `s_currentlyFlashLoaning` will be set to `true` for all tokens. This also means that users who want to take out flashLoans right after an upgrade will be charged the wrong fees. More Importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

**Proof of Concept:**

▶ PoC

```
     function testUpgradedBreaks() public{
      uint256 feeBeforeUpgrade = thunderLoan.getFee();
      vm.startPrank(thunderLoan.owner());
      ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
      thunderLoan.upgradeToAndCall(address(upgraded), "");
      uint256 feeAfterUpgrade = thunderLoan.getFee();
      vm.stopPrank();
      console.log("Fee before upgrade is:", feeBeforeUpgrade);
      console.log("Fee after upgrade is:", feeAfterUpgrade);

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

you can also see the storage layout difference by running `forge inspect ThunderLoanUpgraded storage` and `forge inspect ThunderLoan storage`

Recommended Mitigation:

1. Use a Storage Gap: Reserve a fixed storage gap in the original contract to allow future expansion without interfering with existing variables:
```

```
uint256[50] private __gap;
```

This reserved space ensures storage layout consistency for upgrades.

2. Never Reorder or Remove Storage Variables: Maintain the exact order of storage variables in the upgraded contract as in the original contract. If you need to add new variables, append them to the end of the list.
3. Avoid Replacing Storage Variables with constant: Do not replace storage variables with constant variables in upgraded contracts. Instead:
4. Keep the original variable in place to maintain storage layout integrity. Add the constant variable as a new declaration.