Findings

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

Description:

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

However, the `deposit` function updates this rate without collecting any fees!

```
function deposit(IERC20 token, uint256 amount) external rever
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_P
    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), a
}
```

Impact:

There are several impacts to this bug.

1. The redeem function is blocked, because the protocl thinks the owed tokens is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or 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 Code

Place the following under `ThunderLoanTest.t.sol`

```
function testRedeemAfterFlashLoan() public setAllowedToken
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(t

    vm.startPrank(user);
    tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
    thunderLoan.flashloan(address(mockFlashLoanReceiver),
    vm.stopPrank();

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

Recommended Mitigation:

Remove the incorrectly updated exchange rate line from `deposit`

```
function deposit(IERC20 token, uint256 amount) external rever
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
```

```
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_P
    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), a
}
```

[H-2] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashloaning`, freezing protocol

Description:

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

```
uint256 private s_feePrecision;
uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```
uint256 private s_flashLoanFee; // 0.3% ETH fee
uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variable, breaks the storage location as well

Impact:

Agter the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be

charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

▼ Proof of Code

Place the following into `ThunderLoanTest.t.sol` :

```
import { ThunderLoanUpgraded } from "src/upgradedProtocol/
.
.
.
function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded
    thunderLoan.upgradeToAndCall(address(upgraded), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();

    console.log("Fee before upgrade: %s", feeBeforeUpgrade
    console.log("Fee after upgrade: %s", feeAfterUpgrade);
    assert(feeBeforeUpgrade != feeAfterUpgrade);
}
```

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

Recommended Mitigation:

If you must remove the storage variable, leave it as blanc as to not mess up the storage slot.

```
- uint256 private s_flashLoanFee; // 0.3% ETH fee
- uint256 public constant FEE_PRECISION = 1e18;
+ uint256 private s_blank;
+ uint256 private s_flashLoanFee; // 0.3% ETH fee
+ uint256 public constant FEE_PRECISION = 1e18;
```

[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 receive drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in one transaction:

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA` . They are charged the original fee `feeOne` . 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`

      i) Due to the fact that the way `ThunderLoan` calculates prices based on the `TSwapPool` , this second flash loan is substantially cheaper.

```
function getPriceInWeth(address token) public view returns (u
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getI
@>  return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolToken
}
```

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

Recommended Mitigation:

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