ThunderLoan Audit Report

Version 1.0

Cyfrin.io

August 12, 2025

ThunderLoan Audit Report

Mudassir

August 12, 2025

Prepared by: Mudassir

Lead Auditors: - Mudassir

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

Mudassir 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.

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

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.

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	2
Info	1
Gas Optimizations	1
Total	6

Findings

High

[H-1] Erroneous ThunderLoan::updateExchange 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: Erroneous 'ThunderLoan::updateExchange' 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. 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 even collecting any fees!

Impact: There are several Impacts to this 1: The redeem function gets blocked, because the protocol thinks the amount to be paid is more then its balance. 2: Rewards are incorrectly calculated, leading to way more or way less rewards for liquidity providers. 3: It is now impossible for LP to actually redeem.

Likelihood: High - this will happen everytime as the Deposit function is wrong and severely Impacts the Functionality

```
**Proof of Concept:** 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:

```
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] Storage Collision in C.sol

Description: the upgradeable Contract of ThunderLoan the Variables are out of order, In thunder loan uint256 private s_feePrecision; uint256 private s_flashLoanFee; - but in C.sol uint256 private s_flashLoanFee; uint256 public constant FEE_PRECISION;, This out of order causes this the contract to behave weird and overwrite one variable value with other!!! **Impact:** High - breaks Protocols fee and causes Weirdness

Recommended Mitigation: Donot change the Order of the Variables as u can rewrite one value with another!

```
- uint256 private s_flashLoanFee;
- uint256 public constant FEE_PRECISION

+ uint256 private constant FEE_PRECISION;
+ uint256 public s_flashLoanFee:
```

Low

[L-1] Centralization Risk!

Description: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. Consider limiting the operations of Owner.

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
```

[L-2] Initialize Could be FrontRun!

Description: The Initiazlize function is responsible for Initializing the Contract However Someone else could Front Run this!, or if the Owner forgets to Initialize it can cause Weirdness in the Protocol.

Impact: Low/Medium - Someone else can put their TSwap Address in Oracle and break or Misuse the Protocol!

Recommended Mitigation: Mitigation would be that set in the Deploy Script to automatically set the initialize when creating the Contract so nobody else can initialize after it and u are saved from FrontRunning.

Informational

[I-1] Unchanged Storage Variable should be Immutable or Constant to Save Gas

Description: the S_flashLoanFee storage variable is responsible for flashloanFee but its never changed in the Contract.

Recommended Mitigation:

```
- uint256 private s_flashLoanFee;
+ uint256 private Constant s_flashLoanFee;
```

Gas

[G-1] Unchanged Storage Variable should be Immutable or Constant to Save Gas

Description: the S_flashLoanFee storage variable is responsible for flashloanFee but its never changed in the Contract.

Recommended Mitigation:

```
- uint256 private s_flashLoanFee;
+ uint256 private Constant s_flashLoanFee;
```