Thunder-Loan Audit Report

0xOwain, Cyfrin Updraft

September 2025

Thunder-Loan Audit Report

Version 1.0

0xOwain

September 12, 2025

Protocol Summary

ThunderLoan is a flash loan lending protocol that allows users to borrow tokens without collateral, provided the loan and associated fee are repaid within the same transaction. The protocol integrates with an external AMM (TSwap) to calculate token prices, using pool reserves as an oracle to determine the value of assets during flash loan operations.

The system consists of core contracts for managing deposits, withdrawals, and flash loans, alongside upgradeable logic contracts. Its design emphasizes capital efficiency by pooling liquidity from depositors and enabling borrowers to access instant loans. However, reliance on AMM-based price oracles and the use of upgradeable storage patterns introduces attack surfaces such as oracle manipulation, fee miscalculations, and storage collisions.

Disclaimer

0Owain 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 Likelihood	High	Medium	Low
High	H	H/M	M
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

Repository: Thunder-Loan-Protocol

Commit Hash bc03ba540ab160f8092b7c51f1ed086e179a8087

Contracts in Scope:

- `IFlashLoanReceiver.sol`
- `IPoolFactory.sol`

- `IThunderLoan.sol`
- `ITSwapPool.sol`
- `AssetToken.sol`
- `OracleUpgradeable.sol`
- `ThunderLoan.sol`
- `ThunderLoanUpgraded.sol`

cloc Summary Language files blank comment code ———————————————————————-
Solidity 8 93 255 461 ———————————————————————-
SUM: 8 93 255 461 ———————————————————————-
Out of Scope:
- N/A (only one Solidity contract present in repo)

Severity Criteria

High - Direct loss of funds or permanent lock of assets.
- Anyone can exploit (not just privileged roles).
- Breaks core protocol functionality.

Medium
- Causes significant disruption (DoS, griefing, governance failure).
- Exploitable under some conditions or requires privileged roles.
- Financial loss is possible but limited.

Low
- Minor issues: inefficiencies, gas waste, unclear logic, small inconsistencies.
- Doesn't threaten core security or funds.

Informational / Non-Critical
- Code style, readability, missing comments.
- Best practices (naming conventions, event emissions, input validation improvements).
- No security impact.

Summary of Findings

Severity	Number of issues found
High	2
Medium	1
Low	0
Informational	0
Gas Optimisations	0
Total	**3**

Tools Used

Manual Review: Line-by-line code analysis

Testing: Custom JavaScript tests with Web3

Static Analysis: Slither 0.8.20, Aderyn

High

[H-1] Erroneous `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 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 teh state, without collecting any fees!

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllo
        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);
    }
```

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 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: Remove the updated exchange rate lines from `deposit`.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllo
        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] Mixing up variable location causes storage collisions in `ThunderLoan::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; // 0.3% ETH fee
```

However, the 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 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 will be stored 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 Upgrade: ", feeBeforeUpgrade);
        console2.log("Fee After Upgrade: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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

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. 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.

```
    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.`

I have created a proof of code located in my audit-data folder. It is too large to include here.

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