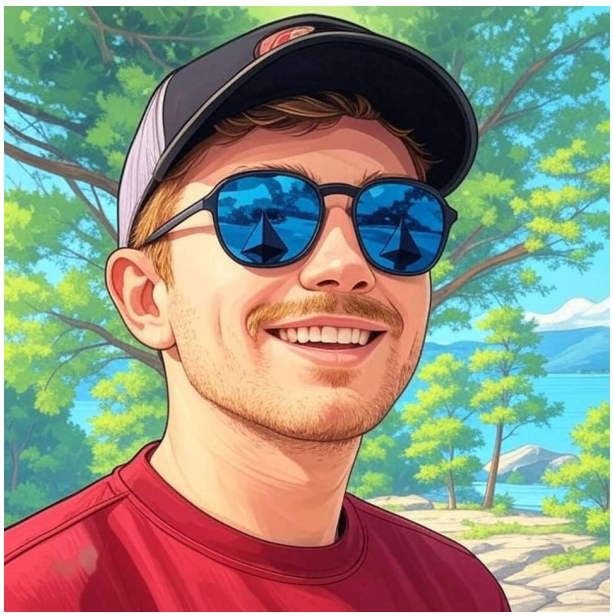

Thunder Loan Audit Report

Version 1.0

Cyfrin.io

August 14, 2025

Thunder Loan Audit Report

Mustby.eth

August 14, 2025

Prepared by: Mustby.eth Lead Auditors: - Matthew Ustby

Table of Contents

- Medium
  - [M-1] Using TSwap as a price oracle leads to price oracle manipulation attacks
  - [M-2] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens.
  - [M-3] `ThunderLoan::deposit` is not compatible with Fee tokens and could be exploited by draining other users funds, making other users lose their deposit and yield.
- Low
  - [L-1] `getCalculatedFee` can be 0.
  - [L-2] `updateFlashLoanFee()` missing event.
  - [L-3] Mathematic operations handled without precision in `getCalculatedFee()` function in `ThunderLoan.sol`.
- Informational
  - [I-1] Poor Test Coverage.
  - [I-2] Not using `_gap50` for future storage collision mitigation.
  - [I-3] Different decimals may cause confusion.
- Gas
  - [G-1] Using bools for storage incurs overhead.
  - [G-2] Using private rather than public for constants saves gas.
  - [G-3] Unnecessary SLOAD when logging new exchange rate.

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 Mustby.eth 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
Likelihood	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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
```

```
7  |   #-- AssetToken.sol
8  |   #-- OracleUpgradeable.sol
9  |   #-- ThunderLoan.sol
10 #-- upgradedProtocol
11     #-- 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

The audit went well. We found 15 vulnerabilities that must be addressed in order to ensure a smoothly running protocol.

Issues found

Severity	Number of Issues Found
High	3
Medium	3
Low	3
Info	3
Gas	3
Total	15

Findings

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 this rate 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   @>      uint256 calculatedFee = getCalculatedFee(token, amount);
8   @>      assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is block, because the protocol thinks that 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 not impossible for LP to redeem

Proof of Code

Place the following into `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 updated 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   -       uint256 calculatedFee = getCalculatedFee(token, amount);
8   -       assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

[H-2] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol.

Description: An attacker can acquire a flash loan and deposit funds directly into the contract using the deposit(), enabling stealing all the funds.

The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds of the AssetContract can be stolen.

Proof of Concept:

To execute the test successfully, please complete the following steps:

1. Place the attack.sol file within the mocks folder.
2. Import the contract in ThunderLoanTest.t.sol.
3. Add testattack() function in ThunderLoanTest.t.sol.
4. Change the setUp() function in ThunderLoanTest.t.sol.

Files not included for this writeup.

Recommended Mitigation: Add a check in deposit() to make it impossible to use it in the same block of the flash loan. For example, registering the block.number in a variable in flashloan() and checking it in deposit().

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

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

```
1     uint256 private s_feePrecision;
2     uint256 private s_flashLoanFee;
```

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

```
1     uint256 private s_flashLoanFee;
2     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 start in the wrong storage slot.

Proof of Concept:

PoC

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

```
1 import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
     ThunderLoanUpgraded.sol";
2 .
```

```
.
.
function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.prank(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);
    }
```

Here is the result of the above test:

```
Logs:
  Fee before upgrade: 3000000000000000
  Fee after upgrade:  1000000000000000000
```

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

Medium

[M-1] Using TSwap as a price oracle leads to price oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM. The price of a token is determined by how many reserves are on either side of the pool. Because of this fact, 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 receive 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 face that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
1      function getPriceInWeth(address token) public view returns (
           uint256) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).
               getPool(token);
3  @>      return ITSwapPool(swapPoolOfToken).
       getPriceOfOnePoolTokenInWeth();
4  }
```

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

The proof of code is located in the `ThunderLoanTest.t.sol` file, as it is too large to include here.

Here are the resulting logs from the `testOracleManipulation` test:

```
1 Logs:
2   Normal Fee is: 296147410319118389
3   Attack Fee is: 214167600932190305
```

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

[M-2] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens.

If the `ThunderLoan::setAllowedToken` function is called with the intention of setting an allowed token to false and thus deleting the assetToken to token mapping; nobody would be able to redeem funds of that token in the 'ThunderLoan::redeem' function and thus have them locked away without access.

[M-3] `ThunderLoan::deposit` is not compatible with Fee tokens and could be exploited by draining other users funds, making other users lose their deposit and yield.

The deposit function does not account the amount for fee tokens, which leads to minting more Asset tokens. These tokens can be used to claim more tokens of underlying asset then it's supposed to be.

Low

[L-1] `getCalculatedFee` can be 0.

`getCalculatedFee` can be as low as 0.

[L-2] `updateFlashLoanFee()` missing event.

`ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` does not emit an event, so it is difficult to track changes in the value `s_flashLoanFee` off-chain.

[L-3] Mathematic operations handled without precision in `getCalculatedFee()` function in `ThunderLoan.sol`.

Informational

[I-1] Poor Test Coverage.

[I-2] Not using `_gap50` for future storage collision mitigation.

[I-3] Different decimals may cause confusion.

For examle, AssetToken has 18, but asset has 6.

Gas

[G-1] Using bools for storage incurs overhead.

[G-2] Using private rather than public for constants saves gas.

[G-3] Unnecessary SLOAD when logging new exchange rate.