Thunder Loan Initial Audit Report

Version 0.1

Cyfrin.io

October 3, 2025

Thunder Loan Audit Report

mhng

October 2, 2025

Thunder Loan Audit Report

Prepared by: mhng Lead Auditors: mhng

- mhng

Table of contents

See table

About mhng

Disclaimer

We are making all efforts to find as many vulnerabilities in the code in the given time period, but we are not holding any responsibilities for the the findings provided in this document. A security audit 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

Audit Details

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

```
2250d81b89aebdd9cb135382e068af8c269e3a4b
```

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.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

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

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info	1
Gas	3
Total	11

Findings

High

[H-1] Mixing up variable location when upgrading the `Thunderloan` contract causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

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

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

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

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

```
1  // You'll need to import `ThunderLoanUpgraded` as well
2  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
3
4  function testUpgradeBreaks() public {
5          uint256 feeBeforeUpgrade = thunderLoan.getFee();
6          vm.startPrank(thunderLoan.owner());
7          ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
8          thunderLoan.upgradeTo(address(upgraded));
9          uint256 feeAfterUpgrade = thunderLoan.getFee();
10
11         assert(feeBeforeUpgrade != feeAfterUpgrade);
12     }
```

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] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution

Description:

Exchange rate for asset token is updated on deposit. This means users can deposit (which will increase exchange rate), and then immediately withdraw more underlying tokens than they deposited.

Documentation: 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!**

Asset tokens gain interest when people take out flash loans with the underlying tokens. In current version of ThunderLoan, exchange rate is also updated when user deposits underlying tokens.

This does not match with documentation and will end up causing exchange rate to increase on deposit.

This will allow anyone who deposits to immediately withdraw and get more tokens back than they deposited. Underlying of any asset token can be completely drained in this manner.

```
    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)
            ;
    }
```

Impact:

Users can deposit and immediately withdraw more funds. Since exchange rate is increased on deposit, they will withdraw more funds then they deposited without any flash loans being taken at all.

Proof of Concept:

```
function testExchangeRateUpdatedOnDeposit() public setAllowedToken {
    tokenA.mint(liquidityProvider, AMOUNT);
    tokenA.mint(user, AMOUNT);

    // deposit some tokenA into ThunderLoan
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    // another user also makes a deposit
    vm.startPrank(user);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);

    // after a deposit, asset token's exchange rate has aleady
       increased
    // this is only supposed to happen when users take flash loans with
        underlying
    assertGt(assetToken.getExchangeRate(), 1 * assetToken.
       EXCHANGE_RATE_PRECISION());

    // now liquidityProvider withdraws and gets more back because
       exchange
    // rate is increased but no flash loans were taken out yet
    // repeatedly doing this could drain all underlying for any asset
       token
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider))
       ;
    vm.stopPrank();

    assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
}
```

Recommended Mitigation:

It is recommended to not update exchange rate on deposits and updated it only when flash loans are taken, as per documentation.

```
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);
}
```

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 { }
```

[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 have 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`.
3. Due to the fact 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   }
```

2. 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.

Low

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

Instances (1):

```
1 File: src/protocol/ThunderLoan.sol
2
3 261:     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):

```
1 File: src/protocol/OracleUpgradeable.sol
2
3 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) |
```

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);
```