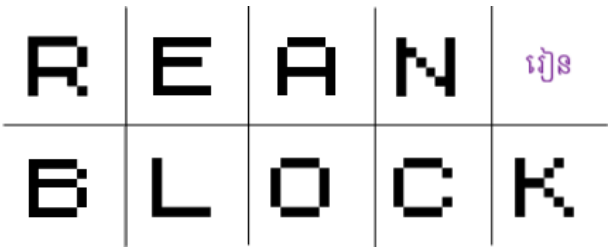

Thunder Loan Protocol Audit Report

Version 1.0

reanblock.com

31st December 2023

Protocol Audit Report

reanblock.com

31st December 2023

Prepared by: Reanblock

Lead Security Researcher: - Darren Jensen

Table of Contents

See table

- Table of Contents
- Introduction
- Overview
- Summary
- Risk Classification
  - Impact
  - Likelihood
  - Actions required by severity level
- Audit Details
  - Scope
  - Roles
  - White Paper / Specifications Document
  - LoC (Lines of Code)
  - SHA-256 File Fingerprints
  - Test Run
  - Test Coverage

  - Linting
  - Slither Security Analysis
  - Aderyn Security Analysis
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
    * [H-2] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution
    * [H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
    * [H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals
  - Medium
    * [M-1] Centralization risk for trusted owners
      · Impact:
      · Contralized owners can brick redemptions by disapproving of a specific token
    * [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks
    * [M-4] Fee on transfer, rebase, etc
  - Low
    * [L-1] Empty Function Body - Consider commenting why
    * [L-2] Initializers could be front-run
    * [L-3] Missing critial event emissions
  - Informational
    * [I-1] Poor Test Coverage
    * [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
  - Gas
    * [GAS-1] Using bools for storage incurs overhead
    * [GAS-2] Using **`private`** rather than **`public`** for constants, saves gas

        * [GAS-3] Unnecessary SLOAD when logging new exchange rate

- Conclusion
- Disclaimer

Introduction

This document outlines the findings for smart contract code review for contracts in Thunder Loan repo at commit SHA 026da6e7 and specifically focuses on the contracts in the `src` folder. All associated test files and deployment scripts were also reviewed as part of the scope of work.

Overview

Project Name	Thunder Loan
Repository	https://github.com/Cyfrin/6-thunder-loan-audit
Commit SHA	`026da6e7`
Documentation	Provided
Methods	Manual review & CLI review (Slither, Aderyn, Solhint)

Summary

The Thunder Loan protocol is meant to do the following:

- Give users a way to create flash loans
- Give liquidity providres a way to earn money off their capital

Liquidity providers can deposit assets into the Thunder Loan protocol and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behaviour with some of the protocol's functionalities that's not so critical.

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions and the cost of the attack is relatively low to the amount of funds that can be stolen or lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - has too many or too unlikely assumptions or requires a huge stake by the attacker with little or no incentive.

Actions required by severity level

- **High** - client must fix the issue.
- **Medium** - client should fix the issue.
- **Low** - client could fix the issue.
- **Informational** - client could consider design/UX related decision
- **Recommendation** - client could have an internal team discussion on whether the recommendations provide any UX or security enhancement and if it is technically and economically feasible to implement the recommendations
- **Gas Findings** - client could consider implementing suggestions for better UX

Audit Details

Scope

In this report the auditor has focused on all contracts in the `src` directory as follows:

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

Roles

The following roles are understood to be part of the protocol:

- 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.

White Paper / Specifications Document

The auditor reviewed the README as provided in the shared repo.

LoC (Lines of Code)

The `cloc` utility was used to determine the lines of code under review. The utility excludes empty lines and comments to leave a count of auditable lines of code in each contract. Since all contracts in scope are under the `src` folder the cloc command was run like so:

```
cloc --md src/**/*.sol
```

The results were as follows:

Language	files	blank	comment	code
Solidity	8	87	207	461
———	———	———	———	———
SUM:	8	87	207	461

SHA-256 File Fingerprints

To generate the SHA-256 fingerprint for all the smart contracts in a directory run:

```
shasum -a 256 src/*.sol
```

Which outputs the following:

```
098c1b0922871c9dc47c08fe0d906910334e3fb68e6694e16a22c334f4064576 interfaces/IFlashLoanReceiver.sol
4564e341e8bb5f76a58b7b11deb9c673b29cc2bd65ab5614b07ee33fc351d340 interfaces/IPoolFactory.sol
1dc4e3241b53a355cf1e0022227c0956bd148ddc8abcd88784c7958dd0274bc2 interfaces/ITSwapPool.sol
84a42e89f39321fc56a2705a55d6ac46885a1766938f52224121224005147795 interfaces/IThunderLoan.sol
01301edc492f4cc1aea26715a76cb7aeae6ae8dfa3bbd3e48f26692a235691a2 protocol/AssetToken.sol
bec13dadcc6bd31ea778f37019f51a8341e86368eab6dff459b9e181ac1bd9ae protocol/OracleUpgradeable.sol
69dcf805a783f2e617ffd639d9a5ae696d191b3e701b2222c83366ac66bbb69f protocol/ThunderLoan.sol
5e146004cca3d6cc81e4181fd618e94a34c85c1c937462a2e0f2d1420ac42470 upgradedProtocol/ThunderLoanUpgraded.sol
```

Test Run

The auditor built the contracts using `forge test` and ran all the tests which are passing (`Test result: ok. 7 passed; 0 failed; 0 skipped; finished in 2.45ms`).

All the contracts were compiled and the test run executed successfully with all tests passing.

Test Coverage

Below shows the output of the test coverage report the contracts in scope. As can be observerd, test coverage should be improved. This issue has been raised in the main section of this report.

File	% Lines	% Statements	% Branches	% Funcs
script/DeployThunderLoan.s.sol	0.00% (0/4)	0.00% (0/5)	100.00% (0/0)	0.00% (0/1)
protocol/AssetToken.sol	80.00% (8/10)	84.62% (11/13)	50.00% (1/2)	83.33% (5/6)
protocol/OracleUpgradeable.sol	100.00% (6/6)	100.00% (9/9)	100.00% (0/0)	80.00% (4/5)
protocol/ThunderLoan.sol	78.12% (50/64)	82.93% (68/82)	43.75% (7/16)	78.57% (11/14)
ThunderLoanUpgraded.sol	1.61% (1/62)	1.25% (1/80)	0.00% (0/16)	7.69% (1/13)
test/auditTests/ProofOfCodes.t.sol	100.00% (11/11)	100.00% (12/12)	50.00% (1/2)	100.00% (1/1)
test/mocks/BuffMockPoolFactory.sol	81.82% (9/11)	87.50% (14/16)	50.00% (1/2)	66.67% (2/3)
test/mocks/BuffMockTSwap.sol	28.79% (19/66)	27.47% (25/91)	10.00% (2/20)	31.58% (6/19)
test/mocks/ERC20Mock.sol	50.00% (1/2)	50.00% (1/2)	100.00% (0/0)	50.00% (1/2)
test/mocks/MockFlashLoanReceiver.sol	81.82% (9/11)	81.82% (9/11)	50.00% (2/4)	100.00% (3/3)
test/mocks/MockPoolFactory.sol	85.71% (6/7)	90.00% (9/10)	50.00% (1/2)	100.00% (2/2)
test/mocks/MockTSwapPool.sol	100.00% (1/1)	100.00% (1/1)	100.00% (0/0)	100.00% (1/1)
test/unit/BaseTest.t.sol	0.00% (0/8)	0.00% (0/8)	100.00% (0/0)	0.00% (0/1)
test/unit/ERC20Mock.sol	0.00% (0/2)	0.00% (0/2)	100.00% (0/0)	0.00% (0/2)
Total	45.66% (121/265)	46.78% (160/342)	23.44% (15/64)	50.68% (37/73)

Linting

Linting is a valuable tool for finding potential issues in smart contract code. It can find stylistic errors, violations of programming conventions, and unsafe constructs in your code. There are many great linters available, such as Solhint. Linting can help find potential problems, even security problems such as re-entrancy vulnerabilities before they become costly mistakes.

After installation, Solhint can be run via the terminal as follows.

```
solhint 'src/**/*.sol'
```

This command will run `solhint` for the contracts in the root of the project directory. The auditor has executed this command and included the output in the Appendix section of this report. The only issue identified by `solhint` was the length of the line in some instances.

I recommend running the solhint linter, either via a Git commit hook or as part of an integration with the developer IDE so that the recommendations can be checked on every code change. NOTE: the above issues are low priority and would not have any impact if not modified.

Slither Security Analysis

Auditor uses Slither (version 0.9.6) static analyzer tool. The slither cli was run against all contracts in scope of the project.

For each of the above contracts the correct version of `solc` using `solc-select` and then run the `slither` command to analyze specific contract under test:

```
solc-select install 0.8.20
solc-select use 0.8.20

slither --checklist --exclude-dependencies .  2>&1 | tee slither-report
    .md
```

The findings were checked and, if appropriate, included in the Executive Summary of this report. The final Slither report is also available as a separate file included with this report.

Aderyn Security Analysis

Auditor uses Aderyn (version 0.0.10) static analyzer tool. The `aderyn` cli was run against all contracts in scope of the project.

Run the `aderyn` command to analyze specific contracts under test as follows:

```
aderyn .
```

The findings were checked and, if appropriate, included in the Executive Summary of this report. The final Aderyn report is also available as a separate file included with this report.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info	1
Gas	2
Total	10

Findings

High

[H-1] Mixing up variable location 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.

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";

function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(upgraded));
        uint256 feeAfterUpgrade = thunderLoan.getFee();

        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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:

```
    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;
```

```
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:

Proof of Concept:

Recommended Mitigation:

[H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

[H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

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):

```
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

[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 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.

```
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 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.

[M-4] Fee on transfer, rebase, etc

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);
```

Conclusion

There are a number of issues reported which are reccommended to be addressed by the protocol developer team.

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s),

assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided “as is” without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.