CyberSaint
SECURITY

Thunder Loan Initial Audit Report

Version 0.1

Sunny Thakur

September 24, 2024

Thunder Loan Audit Report

Thunder Loan Audit Report

Prepared by: Sunny Thakur Lead Auditors:

- *Sunny thakur*

Assisting Auditors:

- *None*

Table of contents

See table

Introduction

The Thunder Loan platform is an innovative decentralized finance (DeFi) application that facilitates peer-to-peer lending and borrowing, leveraging smart contracts on the blockchain. As the adoption of DeFi solutions continues to grow, ensuring the security, integrity, and functionality of these platforms is paramount.

This initial audit report serves as a comprehensive evaluation of the Thunder Loan protocol, focusing on the underlying smart contracts and their interactions. The primary goal of this audit is to identify potential vulnerabilities, assess compliance with best practices in smart contract development, and provide recommendations for enhancing security measures.

Risk Classification

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

Audit Details

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

```
#-- 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.*

Executive Summary

Issues found

Severity	*Number of issues found*
High	*1*
Medium	*2*

Low	*3*
Info	*1*
Gas	*2*
Total	*9*

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:

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

However, the expected 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 postitions of storage variables when working with upgradeable contracts.

Impact: *Afterupgrade,thes_flashLoanFeewillhavethevalueofs_feePrecision. Thismeans 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

RecommendedMitigation: *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 ThunderLoadUpgraded. 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;
```

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: *TheTSwapprotocolisaconstantproductformulabasedAMM(automatedmarketmaker). 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. *DuetothefactthatthewayThunderLoancalculatespricebasedontheTSwapPool this second flash loan is substaintially 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.*

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):

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


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) |
```

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;
```