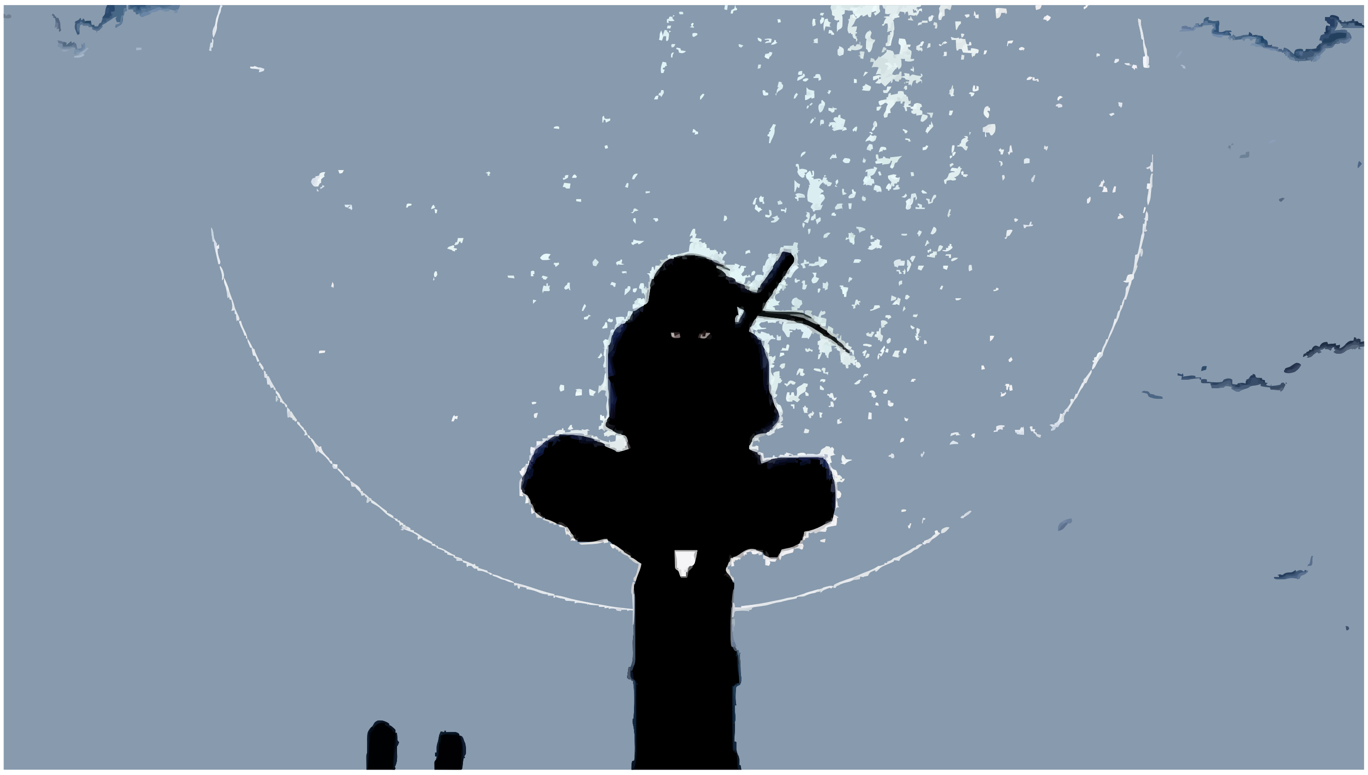

Thunderloan Audit Report

Prepared by: mddragon18

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!

Disclaimer

The mddragon18 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

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

1. Owner: The owner of the protocol who has the power to upgrade the implementation.
2. Liquidity Provider: A user who deposits assets into the protocol to earn interest.
3. User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	2
Low	3
Informational	0

Findings

Highs

[H-1] - Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocol to think it has more fees than it actually does, which block redemption and incorrectly sets the exchange rate.

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens & the underlying tokens.

But the `deposit` function, updates the rate without actually collecting any fees.

```
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:

1. The `redeem` function is blocked because the owed tokens is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers getting way more or less fees.

Proof of Concept:

1. LP deposits.
2. User takes a flashloan.
3. Now LP cannot redeem his tokens.

▶ Code

Place the following in `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);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
amountToBorrow, "");
        vm.stopPrank();

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA,amountToRedeem);
        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the incorrect exchange rate update in the 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] Funds can be stolen using `ThunderLoan::deposit()` instead of paying back flashloan using `ThunderLoan::repay()`

Description: In the `ThunderLoan` contract system after a user takes out a flashloan , he has to repay the loan using the helper `repay()` but this is not the only way, as the `ThunderLoan::flashloan` only checks for the contract balance , the tokens can be directly sent to the contract or `deposit`. If the user deposits the flashloaned funds , the `flashloan` assumes that the user has repaid the loan. This leads to theft of funds as the user can redeem the funds using `redeem()` and hence steal funds of liquidity providers.

Impact: Funds are stolen from liquidity providers and the prootocol.

Proof of Concept:
Steps for the exploit:

1. User takes a flashloan of 50 tokens.
2. User uses deposit function to repay the 50 tokens + fee
3. User is minted asset tokens for depositing.
4. User redeems the asset tokens for the 50 tokens.

Code is included in the `audit-data/h-2.t.sol` , add that code to your test suite.

Recommended Mitigation: Refactor the `repay` function so that it must be called to end the flashloan by adding a `bool` variable of `repaid` and checking if the `repaid` is true in `flashloan`.

[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 Concept:

▶ Code

```
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);
    }
```

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:

```diff
-    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;
```

Mediums

[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 get 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: i. User sells 1000 tokenA, tanking the price. 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();
    }
```

2. 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.

Lows

[L-1]: Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

▶ 6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 239

  ```
      function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
  ```

- Found in src/protocol/ThunderLoan.sol Line: 265

  ```
      function updateFlashLoanFee(uint256 newFee) external onlyOwner {
  ```

- Found in src/protocol/ThunderLoan.sol Line: 293

  ```
      function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

  ```
      function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

  ```
      function updateFlashLoanFee(uint256 newFee) external onlyOwner {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

  ```
      function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
  ```

[L-2]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

▶ 1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 16

  ```
          s_poolFactory = poolFactoryAddress;
  ```

[L-3]: `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

▶ 6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 231

  ```
      function repay(IERC20 token, uint256 amount) public {
  ```

- Found in src/protocol/ThunderLoan.sol Line: 277

  ```
      function getAssetFromToken(IERC20 token) public view returns
  (AssetToken) {
  ```

- Found in src/protocol/ThunderLoan.sol Line: 281

  ```
      function isCurrentlyFlashLoaning(IERC20 token) public view returns
  (bool) {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

  ```
      function repay(IERC20 token, uint256 amount) public {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

  ```
      function getAssetFromToken(IERC20 token) public view returns
  (AssetToken) {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view returns
(bool) {
```